(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
17 May 2001 (17.05.2001)

PCT

(10) International Publication Number
WO 01/35242 A1

(51) International Patent Classification[7]: G06F 15/16, 13/00, G01B 7/00, H04N 7/10, H04J 3/02

(21) International Application Number: PCT/US00/31108

(22) International Filing Date:
13 November 2000 (13.11.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/164,865 12 November 1999 (12.11.1999) US

(71) Applicant *(for all designated States except US)*: **ZEBRA-ZONE, INC.** [US/US]; 53 Issaquah Dock, Sausalito, CA 94965 (US).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: **BARNEA, Gad** [IL/US]; 53 Issaquah Dock, Sausalito, CA 94965 (US).

(74) Agents: **BERNSTEIN, Frank, L.** et al.; Sughrue, Mion, Zinn, MacPeak & Seas, PLLC, Suite 360, 1010 El Camino Real, Menlo Park, CA 94025 (US).

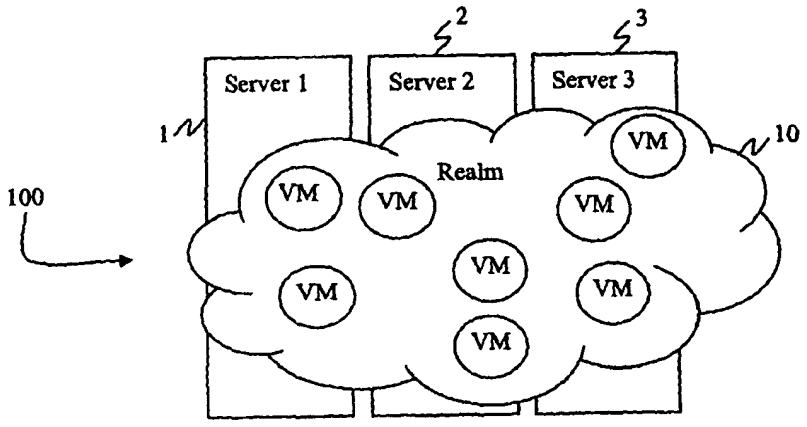
(81) Designated States *(national)*: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *With international search report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: HIGHLY DISTRIBUTED COMPUTER SERVER ARCHITECTURE AND OPERATING SYSTEM



(57) Abstract: A computer server system having a highly distributed architecture (HDA) generally includes a non-hierarchical array of physical machines, each having physical and logical (i.e. virtual) resources (1-3), a network enabling data transmission between and among the physical machines, and program code for allocating and managing system resources (10). The program code for allocating and managing system resources may be in the form of an HDA Server operating system (100) designed to take advantage of the distributed server architecture; advantages of the HDA system (100) may include rapid adaptation to system events and migration of application components among physical and logical resources (1-3). In one exemplary embodiment, a system including an HDA computer server having an HDA Server operating system (100) may serve as a platform for facilitating Internet transactions through use of Adaptive User Interfaces (AUIs) for communication between the HDA system (100) and an external client. Such an HDA-based system provides efficient overall system resource management, excellent fault tolerance characteristics (i.e. stability and reliability), and virtually infinite scalability.

WO 01/35242 A1

HIGHLY DISTRIBUTED COMPUTER SERVER ARCHITECTURE AND OPERATING SYSTEM

The present application claims the benefit of U.S. Provisional Application No. 60/164,865, filed November 12, 1999. The disclosure of that provisional application is incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention is related generally to computer server systems, and more particularly to a computer server system employing a highly distributed architecture and utilizing an adaptive, migrative server-side computer operating system.

2. Description of the Related Art

Conventional computer server systems attempt to address the scalability and bandwidth problems associated with high volume traffic created, for example, by large scale electronic commercial transactions (e-commerce), by providing tier-based server platforms (or "n-tier" architectures). The typical n-tier architecture is built upon a multiplicity of individual and independent servers, each acting autonomously and making use of its own physical and logical resources exclusively. Such configurations generally revolve around central processing hubs (or "tiers"); these tiers host applications or application components under a fixed hardware topology, *i.e.* each component is tied to a specific "physical" location.

Additionally, common and traditional server design methodologies employ threading for internal Session management. In accordance with typical Session management techniques, an n-tier computer server assigns a different thread to each

active Session. While threading may be considered effective for limited uses to a certain extent, the technique is processor intensive, and consequently, slow and cumbersome.

Since each of the independent physical machines in the n-tier server platform is limited to its own individual physical resources, and since threading involves an inefficient tax on overall system resources, the conventional computer server architecture is inadequately adapted to manage heavy system traffic. As a consequence, n-tier computer server platforms, while more flexible than servers built on a single physical machine, still suffer severe limitations in terms of scalability and bandwidth.

SUMMARY OF THE INVENTION

The present invention overcomes the foregoing and other shortcomings of conventional systems by providing software for a computer server having a Highly Distributed Architecture (HDA); in one embodiment, the computer server system of the present invention advantageously utilizes a sophisticated, HDA server-side computer operating system. While one implementation of the HDA Server is generally described herein with reference to an innovative infrastructure enabling Business Service Providers (BSPs) to provide value-added services to their customers on the World Wide Web (Web), it is within the scope and contemplation of the invention to employ such an HDA Server in other contexts for other applications, such as intranet connectivity, home networking, pervasive computing, and so forth.

In accordance with one aspect of the present invention, a computer server having a highly distributed architecture generally includes a plurality of physical machines (PMs), each having physical and logical resources, a network enabling data

transmission between and among the PMs, and program code for managing system resources. As noted above, the program code for managing system resources may be in the form of an HDA operating system designed to take advantage of the HDA Server architecture.

The benefits of implementing such an HDA-based system include efficient overall system resource management, excellent fault tolerance characteristics (*i.e.* stability and reliability), and virtually infinite scalability as well as ease of maintenance and application deployment.

In accordance with another aspect of the invention, a system including an HDA computer server is employed for serving as a platform for facilitating Internet transactions. In one embodiment, for example, a system employing the HDA Server of the present invention may provide "SoftSpot" technology. A SoftSpot is an Adaptive User Interface (AUI). SoftSpots may exist independently of a host user interface, but may be able to integrate into the host interface. SoftSpots may be embedded in a Web page, but they may also function independently, for example, in a cellular telephone Wireless Application Protocol (WAP) interface or in a Voice-driven interface.

The above-mentioned and other attendant advantages of the present invention will become more apparent upon examination of the following detailed description of the preferred embodiments thereof with reference to the attached drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a simplified block diagram of one embodiment of a computer server employing a Highly Distributed Architecture (HDA).

Figure 2 is a simplified block diagram of one embodiment of an HDA computer server connected to a network.

Figure 3 is a simplified block diagram of one embodiment of the Registry component employed by an HDA Server.

Figure 4 is a simplified flow chart illustrating one embodiment of the operation of the Registry component of an HDA Server.

Figure 5 is a simplified flow chart illustrating one embodiment of the data flow through the Adaptive Messaging Services component.

Figure 6 is a simplified flow chart illustrating one embodiment of the procedures for runtime activation and initialization of Application software employed by an HDA Server.

Figure 7 is a simplified block diagram illustrating one embodiment of an HDA Server operating system for use in conjunction with an HDA Server.

Figures 8A and 8B illustrate an XML representation of one embodiment of a TopologyDescriptor employed by an HDA Server OS.

Figure 9A is an example of one embodiment of a Migration Request event.

Figure 9B is a simplified flow chart illustrating one embodiment of a Migration employed by an HDA Server OS.

Figure 10 illustrates one embodiment of an svcs.xml file which may govern the boot process for an HDA system.

Figure 11 is a simplified flow chart illustrating one embodiment of an HDA system boot process.

Figure 12 is a simplified block diagram illustrating one embodiment of a Server Gateway which may be employed by an HDA Server.

Figure 13 is a simplified flow chart illustrating the operation of one embodiment of a Server Gateway employed in conjunction with an HDA Server.

Figure 14 is a simplified block diagram illustrating the operation of one embodiment of database access which may be employed by an HDA Server.

Figure 15 is a simplified block diagram illustrating the operation of another embodiment of database access which may be employed by an HDA Server.

Figure 16 is a simplified flow chart illustrating one embodiment of the life cycle of a SoftSpot which may interact with an HDA Server.

Figure 17 is an illustration of one embodiment of a SoftSpot descriptor.

Figures 18A and 18B are an illustration of one embodiment of a Uniform SoftSpot Descriptor file.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Turning now to the drawings, Figure 1 is a simplified block diagram of one embodiment of a computer server employing a Highly Distributed Architecture (HDA) and an HDA Server operating system. In the exemplary embodiment illustrated in Fig. 1, the HDA Server 100 having a distributed architecture in accordance with the present invention takes advantage of both physical and logical resources, irrespective of physical "location." Computer servers 1-3 represent the physical resources available to HDA Server 100; servers 1-3 are physical machines (PMs) comprising electrical and

electromechanical components, such as conventional file servers, application servers, or mini-mainframes, for example.

The Realm 10 may function as a logical container for various elements of HDA Server 100, such as an HDA Server operating system (OS) and the applications and data resources it serves. The various system elements contained in Realm 10 are not bounded by physical location, and may utilize available system resources at any of servers 1-3, either individually or in combination. Realm 10 may contain one or more virtual machines (VMs), *i.e.* logical resources supported by PMs. As described in detail below, the number of VMs, as well as the "location" (*i.e.* the physical resources utilized) of each VM, may change dynamically over time in accordance with system demands. Similarly, the location or physical resources utilized by Realm 10 may also change dynamically over time.

As noted briefly above, servers 1-3 may be conventional file servers, application servers, mini-mainframe computers, or other types of PMs known in the art. The functionality of HDA Server 100 and the HDA Server OS is not dependent upon the nature or the specific arrangement of the components employed by servers 1-3; for example, chip sets, bus architectures, memory configurations, and the like, may vary from server 1 to server 3 without affecting the operation of HDA Server 100. Invocation and general management of VMs is known in the relevant art.

The general embodiment of HDA Server 100 depicted in Fig. 1 is illustrated by way of introduction and example only, and not by way of limitation. The HDA system architecture of the present invention may support a desired number of VMs in more than one Realm, such as Realm 10, across a desired number of PMs, such as servers 1-3. In

particular, though Fig. 1 only illustrates one Realm (Realm 10) and three servers (servers 1-3), it may be desirable to provide an HDA system with more physical resources, *i.e.* more than three PMs, and more Realms than are shown in Fig. 1. By way of example, one embodiment of the present invention operates with three Realms, as discussed below with reference to Fig. 2. It will also be appreciated that a single Realm may span any number of PMs and may incorporate any number of VMs.

Figure 2 is a simplified block diagram of one embodiment of an HDA computer server employing an HDA computer operating system and connected to a network. The HDA Server 200 generally corresponds to the distributed HDA Server 100 in Fig. 1; it will be appreciated that the elements of HDA Server 200 described below may be distributed across, and utilize the physical resources of, one or more PMs as set forth above with reference to Fig. 1. In the embodiment of Fig. 2, HDA Server 200 is generally constituted by the following components: Realms 210, 220, and 230; a Registry component 240; an Adaptive Messaging Services component 250; and a Transaction Services Manager component 260. A Server Gateway 270 and a SoftSpot Gateway 280 may combine to enable data communication between HDA Server 200 and a network 999 such as the Internet, for example. As noted above, the foregoing elements, and parts thereof, may be contained in one or more VMs distributed across multiple PMs.

As described in detail below, a Data Layer Realm 220 may have access to data records maintained in one or more databases 290. An accessible database 290 may be maintained at HDA Server 200, or may be maintained at one or more remote servers.

When maintained by HDA Server 200, database 290 may be distributed across multiple VMs at different PMs in a similar fashion as the server elements described above.

The arrangement of HDA Server 200 is representative of an exemplary HDA. The architecture and operation of HDA Server 200 are very different from tier-based server platforms. The arrangement (of PMs, VMs, and Realms 210, 220, and 230) which forms the foundation of the architecture for HDA Server 200 may be viewed as a medium through which objects may "flow" or migrate between VMs (and PMs). In this context, the term "object" refers to computer programming code components created in accordance with object-oriented programming techniques.

Those of ordinary skill in the art will appreciate that HDA Server 200 is more flexible than n-tier architectures. Whereas conventional n-tier servers employ a hierarchical structure of independent physical machines, generally involving at least one "main" or "master" PM operating at a level or tier above the other PMs, such a multi-level or multi-tier hierarchy does not exist in the inventive HDA Server 200. Rather, the physical resources of HDA Server 200 may generally be constituted by a non-hierarchical array of cooperating PMs across which logical resources and objects may be distributed. The term "non-hierarchical" in this context refers to the fact that HDA Server 200 may be considered to be a single tier, or at least not requiring that the PMs be linked in a multiplicity of levels or tiers.

Additionally, HDA Server 200 may advantageously employ a "publish/subscribe" model of data transmission, where data may be published and subsequently automatically routed to a particular component which has registered as a subscriber to these data. In contrast with most tier-based systems, the actual data

expected (*i.e.* "subscribed" to) by a component does not need to be hardcoded into the object itself; additionally, the object is not tied to any "physical" location.

REALMS

As noted briefly above, Realms 210, 220, and 230 are logical containers for the various components of HDA Server 200. In effect, a Realm may be viewed as an "internal server" of sorts, performing specific functions and serving particular needs within HDA Server 200. For example, a Data Layer Realm 220 may perform such an internal server function, for example, serving data in an Extensible Markup Language (XML) format which other applications (served by HDA Server 200) can understand. As illustrated in Fig. 1 and discussed above, Realms 210, 220, and 230 may contain VMs. The number of VMs in any particular Realm may change dynamically over time. Also, the "location" of a Realm and its VMs (*i.e.* the physical resources utilized by each) may advantageously change dynamically over time.

Each Realm 210, 200, and 230 manages a set of VMs and PMs. One Realm in particular (Operating System (OS) Realm 210) is unique in that may actually manage other Realms. The OS Realm 210 represents the HDA Server operating system which manages all the VMs and physical hardware resources available to HDA Server 200. Other Realms 220 and 230 may communicate with OS Realm 210. In the embodiment illustrated in Fig. 2, OS Realm 210 may manage the low-level resources, while the "higher-level" Realms 220 and 230 may manage the application logic components.

As will be addressed in detail below, Realms 220 and 230 may be assigned management of only one type of component directly, namely, Services (such as denoted as 250 and 260 in Fig. 2). In this embodiment, the functions of Realms 220 and 230

may involve keeping Services alive, spawning Services at boot time or when new instances are needed, and generally implementing any directive given by OS Realm 210. OS Realm 210 may follow the same rules as other Realms 220 and 230; in other words, the OS resource management may be managed at the Services level as set forth below.

In operation, the object-oriented program code for Plugins and Applications (both of which are discussed below) may be "contained" in Services, whereas the Services themselves may be "contained" in Realms. As noted above, a Realm employs a set of VMs allocated to that particular Realm. Using a file system metaphor to picture these logical organizational entities: a Realm may be considered a volume (drive); a Service may be considered a folder; and a Plugin may be considered a file.

Still referring to Fig. 2, the Application Realm 230 may maintain all the Applications which are installed in HDA Server 200, *i.e.* the Applications which may be served. The term "installed" denotes Applications which were installed by installation software code or programming scripts provided by OS Realm 210. The term "AppInstaller" (discussed below) is used to denote this installation software. In one embodiment, Application Realm 230 may only manage the Services that govern Applications. In operation, Application Realm 230 may be the most dynamic Realm in HDA Server 200. Consequently, resource allocation and management in Application Realm 230 may be substantially more active than with other Realms; this does not imply any architectural issues, but rather system resource deployment issues. In accordance with expected system demands, Application Realm 230 may receive more physical and virtual resources at deployment time than the other Realms.

It will be appreciated from the following discussion that other Realms may be just as "active" in terms of processes as Application Realm 230, but since these other Realms are less "dynamic" in terms of resource demands, caching and other optimization algorithms may be used to ease the resource management burden on the other Realms in a way which may not be applicable to Application Realm 230.

In one embodiment, Application Realm 230 may have the highest priority with the OS employed by HDA Server 200; the OS may monitor processes taking place in Application Realm 230 intensively. Additionally, Application Realm 230 may generally be the last Realm to be booted by the Launcher (discussed below). That is, Application Realm 230 may advantageously be booted after both the Data Layer Realm 220 and OS Realm 210 have booted successfully.

The Data Layer Realm 220 may maintain Services which deal with querying and updating various types and quantities of data sources. By way of example, Data Layer Realm 220 may handle Java Database Connectivity (JDBC) Services, Common Object Request Broker Architecture (CORBA) Services, as well as Simple Object Access Protocol (SOAP) Services. In operation, Data Layer Realm 220 may be much less dynamic than Application Realm 230, but not less active. New types of data Services may be installed dynamically into Data Layer Realm 220. In one embodiment, Data Layer Realm 220 may be the second Realm to be booted by the Launcher (*i.e.* subsequent to successful boot of OS Realm 210).

As a result of its being less dynamic than Application Realm 230, Data Layer Realm 220 may employ many optimizations (such as advanced caching, templating, and pooling, for example) which are relatively easier to implement in Data Layer Realm 220

due to its fairly "static" nature; these optimizations may dramatically speed up performance of processes in Data Layer Realm 220.

It should be appreciated that, in one embodiment, it may be desirable to configure Data Layer Realm 220 such that the input to the Data Layer from the other elements of HDA Server 200 may be exclusively in a single markup language, XML or Document Object Model (DOM) format, for example; output from the Data Layer to HDA Server 200 may be in the same markup. Input and output format to and from the various data sources, however, may depend upon the specific data source. In other words, all data communications which are internal to HDA Server 200 may be in a single desired markup, while external communication markup may be dictated by the external source.

As noted briefly above, OS Realm 210 may encapsulate the server-side HDA computer OS which governs low-level operation of HDA Server 200. As with any operating system, the OS employed by HDA Server 200 may be assigned the task of managing resources and processes while providing software "drivers" for use by applications. One major distinction between the OS employed by HDA Server 200 and conventional systems is the fact that the OS of the present invention may advantageously be designed as natively distributed; that is, in one embodiment, the OS is engineered from the bottom up to be a full server-side distributed OS for use with HDA Server 200.

In one exemplary embodiment, OS Realm 210 may maintain a set of Services which govern various OS functions; as noted above, OS Realm 210 may be different from other Realms in this sense, because it may be operating entirely "offline." This

means that OS Realm 210 may not be affected by user Sessions (at least not directly) as described below. Those of skill in the art will appreciate from the foregoing that OS Realm 210 may preferably be the first Realm to be booted by the Launcher.

SERVICES

By way of further introduction, Services are the actual processes which drive the operation of HDA Server 200; these Services may generally be managed by software code provided in each respective Realm, as discussed above. In one embodiment of HDA Server 200, however, Services may be oblivious to the operations carried out in the various Realms, and may not interact with Realms 210, 220, and 230 directly. In this embodiment, two kinds of Services may be employed by HDA Server 200: SessionAwareServices; and DaemonServices.

Most Services may be SessionAware, or Session dependent. It may be desirable, however, that all Services in OS Realm 210 be Daemon Services, which may be viewed as maintaining an "eternal Session."

Services in this embodiment may be transient software components; Services in Data Layer Realm 220 and Application Realm 230, for example, may never need to be continually maintained, since they may not directly hold any critical data. Services may easily be selectively spawned or terminated at any time by the associated Realm. The main functionality of Services is to act as a container for Plugins and to manage Messaging safely and independently. All Services may be, in fact, identical software components; that is, the Services may differ only with respect to the components each contains. These components may advantageously be assigned to each Service dynamically. Every Service may be contained in a managerial and administrative

object, referred to herein as ServiceManager, which essentially keeps track of the various instances of the same Service. Each Realm 210, 220, and 230 may create its own ServiceManager for internal use and organization.

SessionAware Services may constantly manage Services for a specific set of Sessions. Contrary to all common and conventional design methodologies, internal Session management in HDA Server 200 may not involve using different threads for each active Session. Rather, the Service may maintain a list of DOM documents, each of which may contain data relating to a specific Session. All communication between Services and other entities may be accomplished through the Messaging System (discussed below) which may deliver DOM documents stamped with an associated SessionID. This is a much faster and more efficient (*i.e.* less resource-consuming) approach than the common method of using Java threads, for example.

Daemon Services may constantly perform a task for a single "eternal" Session. Daemon Services may be considered to be semi-mission-critical; as such, these particular Services may be managed differently by the associated Realm. For example, in the case where one of these Services fails for any reason, spawning a new instance of such a failed Service may receive the highest priority. Again, as with SessionAware Services, there is no issue of persisting data, and consequently, spawning a new Daemon Service may be virtually instantaneous.

REGISTRY & ADAPTIVE MESSAGING SERVICES

The simplified block diagram of Fig. 2 illustrates a Registry component as element 240. Registry 240 may control and maintain the publish/subscribe capabilities of HDA Server 200. Since HDA Server 200 utilizes a loosely coupled architecture,

inter-component interaction need not be hardcoded; in fact, in one desirable embodiment, the various components are not aware of any other entity except Registry 240. In this embodiment, Registry 240 may be implemented around a replicated Lightweight Directory Access Protocol (LDAP) server. LDAP is fast and efficient, and as a result, assists HDA Server 200 in allocating system resources quickly and economically. Such an arrangement also allows HDA Server 200 to make use of the advanced security features of the Java Naming and Directory Interface (JNDI) which communicates with LDAP. Many robust and mission-critical LDAP servers are known in the art.

Figure 3 is a simplified block diagram of one embodiment of a Registry component employed by an HDA Server. As will be appreciated from examination of Fig. 3, the Registry component 240 of Fig. 2 may be generally constituted by four components: Registry 340; Antenna 343; Multicaster 344; and Directory Interface 341. As will be described below, Antenna 343 may serve to receive published events and to convey or to direct those published events to Registry 340; in that regard, the term "antenna" is intended to be descriptive of the function, but not necessarily the structure, of Antenna 343.

As illustrated in Fig. 3, Registry 340 may be the only access point to the LDAP server 349; this access may be through Directory Interface 341 (JNDI interface wrapper). Registry 340 may store all published events and a list of all subscribers for each event in LDAP server 349.

When Services get initialized (at boot time or any other time during the lifetime of the system), these Services may bind themselves to Registry 340; this may be done

through an Object Request Broker (ORB), such as ORB 342. ORBs are well known in the art. Such binding may also be reflected in LDAP server 349. If for any reason, Registry 340 fails, the entire failed Registry 340 and its data contents may easily and rapidly be reconstructed by the OS from the data retained in LDAP server 349.

In operation, Registry 340 may function as a sort of "central nervous system" for the HDA system of the present invention. Registry 340 may act as an interface to a private set of objects which provide runtime data management Services.

By way of example, Registry 340 may aggregate three components shown in Fig. 3 (*i.e.* Antenna 343, Multicaster 344, and Directory Interface 341) and may manage data communication between them. In this embodiment, Registry 340 may receive event names from Antenna 343 and transfer those event names to Directory Interface 341 for comparison with a list of subscribers maintained in LDAP server 349; the list of subscribers, in turn, may then be transmitted to Multicaster 344 for subsequent transmission of data to all subscribers. In addition, Registry 340 may also receive binding information from ORB 342 or Multicaster 344 and transmit that information to LDAP server 349 through Directory Interface 341.

As noted briefly above, Antenna 343 may be a "well-known" recipient of published events, *i.e.* Antenna 343 may be a subscriber to every system event. In this embodiment, events may be published by any component bound to Registry 340 (typically a Service or a Transmitter, discussed below) through Antenna 343. That is, when Antenna 343 receives an event, it may extract the event's name and transfer it to Registry 340. If there is at least one subscriber to the event, Registry 340 may add the event's data to the Message object to be sent through Multicaster 344 to subscribing

components. However, any component (even those which are not bound to Registry 340) may be able to use the Service or the Transmitter to publish events by proxy; the Server Gateway 270 in Fig. 2 and discussed below, is a subclass of a Transmitter. In one embodiment of an HDA system connected to an external network, such as the Internet, for example, most event publishing may be issued by Sever Gateway 270.

It will be appreciated that publish/subscribe models are known in the art; in traditional systems, however, each message remains constant from the moment it is published through the moment it is received by the subscriber. The present implementation, however, advantageously allows Message objects to be updated with new data as described above.

Figure 4 is a simplified flow chart illustrating one embodiment of the operation of a Registry component of an HDA Server. At block 401, an event may be published, for example, by the Server Gateway 270 illustrated in Fig. 2. The event arrives at Antenna 343 where Antenna 343 may separate the event's NAME from its DATA, and delegate the event's NAME to the Registry 340 (this is illustrated at blocks 402-404). As indicated at block 405, Registry 340 may query LDAP Server 349 though Directory Interface 341 to obtain a list of subscribers to the event; if at least one subscriber is found, Registry 340 retrieves the event's DATA and adds it to the Message object (block 406) which is to be multicast to all subscribers; Registry 340 then may pass this Message object to Multicaster 344 at block 407. Finally, Multicaster 344 transmits the full event (*i.e.* the same Message object that was received by Antenna 343) to all subscribers simultaneously, as noted at block 408.

It will be appreciated that the foregoing sequence of events is provided by way of example only, and that other embodiments or configurations of the elements shown in Fig. 3 may be desirable. For example, Antenna 343 and Multicaster 344 may be combined into a single, dual-function transceiver element.

Figure 5 is a simplified flow chart illustrating one embodiment of the data flow through the Adaptive Messaging Services component. As will be appreciated from examination of Fig. 5, the Adaptive Messaging Services component 250 of Fig. 2 may be generally constituted by computer programming code generating data Message objects to be distributed throughout HDA Server 200; this programming code is represented by the MessageFactory 551 in Fig. 5. The Server Gateway 270 of Fig. 2 is illustrated in Fig. 5 as communicating with Session 571.

Message objects in the HDA Server system may generally be considered containers and transport vehicles for XML/DOM documents. A Message object may be created by any Transmitter through MessageFactory 551. Message objects are moveable objects, *i.e.* they are ORB agents which the Registry and Transmitters may send to other objects. Messages may communicate with objects which implement an appropriate interface, for example, which enables the objects to interpret Message objects from MessageFactory 551; this interface is referred to as a MessageListener interface.

By way of example, an incoming message from a remote location, a Web-based server, for example, may contain XML data as shown in Fig. 5. This XML data arrives at Server Gateway 270, which may transmit the received XML data to Session 571. In this embodiment, Session 571 may translate the XML to DOM (creating a DOM

document); Session 571 is preferably a Transmitter object, *i.e.* Session 571 may call a function of MessageFactory 551 to create a new Message object.

It will be appreciated from examination of Fig. 5 that the output of MessageFactory 551 may be a proprietary markup language, or any other desirable data format such as XML or DOM, for example, at least for data communications internal to the HDA Server of the present invention. It is within the scope and contemplation of the present invention to make use of any and all suitable markup languages for internal data transfer.

TRANSACTION SERVICES

Returning now to Fig. 2, the Transaction Services Manager component 260 may maintain and manage various transactional interactions between the several components of HDA Server 200. The following discussion provides an introduction to Transactions.

In one embodiment, all Transactions may share the following properties (referred to in the art as ACID): Atomicity; Consistency; Isolation; and Durability.

Atomicity: any indivisible operation (one which will either complete fully or not at all) is said to be atomic.

Consistency: A Transaction must transition persistent data from one consistent state to another. If a failure occurs during processing, the data must be restored to the state it was in prior to the incomplete or failed Transaction.

Isolation: Transactions should not affect each other. A Transaction in progress must be isolated from other Transactions. Although several Transactions may run

concurrently, it should appear to each that all the others completed before or after it; all such concurrent Transactions must effectively end in sequential order.

Durability: Once a Transaction has successfully been completed, state changes committed by that Transaction must be durable and persistent, despite any failures which may occur afterwards.

A Transaction may end in one of two ways: a "commit," which represents the successful execution of each step in the Transaction; or a "rollback," which guarantees that none of the steps are executed due to an error in one of those steps. In one embodiment, the HDA system of the present invention may employ Java Transaction API (JTA) to control Transactions. Transactions may be critical to several operations in the HDA Server.

With respect to the Registry component, for example, the persistence of subscriber information in the LDAP server is critical, and that action is Transactional; maintaining this persistence may be the responsibility of the Directory Interface. Some database updates may be critical, and are therefore Transactional. As with any server system, some processes in the OS Realm are also Transactional; for example, booting the system and the migrative process of various components (both of which are discussed below in detail) may both be considered Transactional.

PLUGINS

Plugins are components which may be "plugged in" to an HDA system configured in accordance with the present invention. The loosely coupled architecture of an HDA system allows a Plugin to be installed while the HDA Server is up and running. On the administrative level, all such Plugin components may be installed, for

example, through a management console or other installation interface (referred to as a Management Console, or MC), in such a way that the HDA Server OS may load each Plugin's respective program elements and attach each to the proper Services in the correct Realm. Plugins may also be removed and modified without any effect on system performance.

On the system level, all Plugins may be installed through the use of a computer programming script or installation software code providing a suitable interface, called a PluginInstaller interface. Plugins may be installed into the Application Realm, and a suitable installer program may be, for example, AppInstaller discussed above. An actual Plugin will typically extend the abstract Plugin class and optionally implement a SessionAwarePlugin interface. A Plugin may be treated by the system as a resource capable of handling requests via its implementation of a RequestHandler interface. A SessionAware Plugin may keep state between consecutive calls to handle request.

In general, Plugins and their implementation are known in the art. The following example provides a detailed description of a Plugin installation procedure; this description assumes that the Plugin to be installed is already in a recognized location, for example, PLUGIN_PATH.

A Plugin manager software code or script (for example, program code referred to as a PluginManager object) in the OS Realm may un-jar (or uncompress) the jar file containing the Plugin and store the uncompressed data into a temporary directory created for this particular Plugin. The PluginManager may first read security information. In instances where security is restricted, the Plugin may be assigned certain restrictions. For example, the restricted Plugin may not be able to read or to

write files, or it may be restricted with respect to memory allocation. In one embodiment, the default security profile for all Plugins may be <fully-restricted>.

The PluginManager may then transfer a Plugin descriptor to the installer specified in the <installer> element of the Plugin descriptor (such an <installer> element may not be required, since, for example, a default installer may be provided for the Application Realm). For the purposes of this example, the default installer software is AppInstaller.

The AppInstaller object may publish an event, for example a <new-plugin> event, to alert other elements of the HDA Server that a new Plugin is being made available. Preferably, a Realm management software script (in this example, referred to as RealmManager) keeps handles to all Realms and is further subscribed to the <new-plugin> event. By virtue of being subscribed to the <new-plugin> event, RealmManager may be apprised of the publication of the event by AppInstaller. In this embodiment, RealmManager may read the DOM document of the event (the <realm> element of the Plugin descriptor), and may return a handle to the Application Realm.

The AppInstaller may then move to the Application Realm (the AppInstaller is an ORB agent and may move to any known address space). In conjunction with the move, AppInstaller may bring copies of all the necessary resources (the Plugin jar file and any resource defined by the <resource> element, for example). The Application Realm may then be notified of the arrival of AppInstaller. (In this regard, Application Realm may implement an interface through which it may be notified of the arrival of, and communicate with, the AppInstaller).

AppInstaller may then transfer the desired <plugin> node from the Plugin descriptor to the Application Realm's notifyPluginManager(Node, String) program script, thus transferring control to the Application Realm. Application Realm may call createPluginDescriptor(Node, String) on PluginManager and transfer the <plugin> node as the DOM document.

PluginManager may then load the resources specified in the <resources> element. For jar files, for example, PluginManager may use java.util.jar.JarFile to load the classes of the "required" jar files. Additionally, a mechanism may be provided for "lazy" or "as needed" initialization of resources.

PluginManager may attach the Plugin to the specified Service type (for example, a SessionAware Service). Additionally, PluginManager may call the TopologyManager (part of the HDA Server OS, discussed below) to find the best VM (*e.g.* the VM with the most available resource capacity) in the Application Realm for installation of the Plugin.

PluginManager may load and initialize the Plugin. The Plugin then may finish initialization by loading and initializing its Formatter and Transmitter. The Plugin may then call PluginManager's function devoted to updating the Registry; as a result, PluginManager may notify the Registry, transmitting all the data provided by the Plugin.

When registration with the Registry returns successfully, the Plugin's internal state may be set to running, at which time the Plugin is open to receive incoming events. It should be noted that the foregoing example of Plugin installation is a Transactional process, *i.e.* guaranteed either to succeed without a single fault or to fail completely.

APPLICATIONS

Applications are entities which form the "business logic" of the HDA Server. Those of skill in the art will appreciate that an Application may generally be composed of a set of Java objects and XML descriptors. In one embodiment of an HDA system, all Applications may share the same contract of mandatory and optional objects and a common set of descriptors. The contract is designed to allow Applications to be installed automatically, deployed, and activated in the system simply by placing a compressed jar file containing all the required resources in a particular recognizable location. In this embodiment, all Application resources may be found in a particular path, *e.g.* APPLICATION_PATH, which may be managed by the AppInstaller (a component of the OS Realm as described above). The contract may also allow different Applications to communicate with each other and to share data.

By way of example, the life cycle of an Application may begin upon loading the jar file that contains all the resources for the particular Application to the APPLICATION_PATH. This loading may be accomplished through an MC as described above with reference to Plugins in general. The MC may be command-line based, for example. Installation program code, such as AppInstaller, may learn of the existence of the new Application (discussed in detail below) and subsequently decompress the jar file into all relevant locations across all relevant address spaces. The deployment topology may be defined in the XML descriptor files packaged with the Application. After deployment has been successfully executed, AppInstaller may notify the Registry that a new Application is available. In effect, AppInstaller publishes an event indicative of the installation, for example, a "NEW_APPLICATION" event, to

the Registry. The Registry, then, may notify the subscribers to this event (an example of which may be the Application Realm's Receiver Service, described below) that this event has occurred. The process may be similar every time the system boots.

Figure 6 is a simplified flow chart illustrating one embodiment of the procedures for runtime activation and initialization of Application software employed by an HDA Server. It should be noted that the following example only summarizes the events which may occur after the Receiver Service has been notified of the existence of a new Application, and is provided by way of background only, and not by way of limitation.

At block 601, the Receiver gets a notification from the Registry that an event, for example, NEW_APPLICATION, has occurred, at which point the Receiver may request the DOM document describing the Application from the originator of the event, for example, AppInstaller (this request is reflected in block 602). The Receiver may create an Application Descriptor (WADescriptor) object containing a descriptive DOM document at block 603, and may subsequently transfer the WADescriptor to an Application Manager (WAManager) at block 604. Finally, the WAManager may initialize the "main" class for an Application object (block 605) before the Application object begins its own initialization (block 606).

The first step in the initialization process involves the Application (subclass) object reading the WADescriptor and initializing its various components (block 607). Sub-components (contained) are initialized as shown at block 608. Those of skill in the art will appreciate that the Application object may manage any number of other objects. This object aggregation deals with application behavior and may preferably be exclusively Java based. Additionally, the Application's Formatter object is initialized at

block 609 (the Formatter object attached to the Application may generally be a "contractual" container for the Application's formatting data which is contained in its DOM document).

The Application also preferably contains Transmitter and Receiver objects. These objects may contain and maintain the data manipulation DOM document for the associated Application object. These components may form the part of the Application which may engage in system-wide "publish/subscribe" through the Registry as described above. Initialization of the Application's Transmitter and Receiver objects is illustrated at block 610.

The following describes how an incoming event may be processed by one embodiment of an exemplary Application, CustomerService.

An event, for example, <user-event> containing <customer_inquiry> data, arrives at the Server Gateway. In this example, the event may arrive from a "known" customer, *i.e.* one who has a known customer ID. For data communication internal to the HDA Server, the event's data may be contained in a DOM document (*i.e.* the Server Gateway may translate the XML received from the Hypertext Transfer Protocol (HTTP) Stream to DOM as described above). The XML in such an instance may resemble the following:

```
<user-event format=wap>
    <customer_inquiry>
        <customer_id>1cb4af59</customer_id>
        <inquiry_text>I would like to get a quote on buying 34 units of
                your product at corporate rate, Thanks. Jim
        </inquiry_text>
```

```
            <priority>medium</priority>
        </customer_inquiry>
    </user-event>
```

The Server Gateway may open a new Session or join an existing Session; the Session format may be WAP, for example. As described above, the Session may publish the event, stamped with a unique SessionID, to the Registry such that the Registry may multicast the event to all subscribers.

In this example, a CustomerService object may be one of the subscribers (or the only subscriber), so it will be notified of the event. CustomerService, a subclass of Application, has a Transmitter object attached to it as described above with reference to Fig. 6. It will be appreciated that CustomerService previously had to use the its Transmitter object in order to subscribe to an event (in this case the <customer_inquiry> event). CustomerService may delegate the event's DOM document to the appropriate EventListener.

As noted briefly above, an appropriate EventListener may be capable of "reading" the event's DOM document; the EventListener may then process the data contained in the DOM document according to the logic defined for this particular event.

Further, the EventListener may "publish" an internal event, for example <QueryEvent>, to which the Data Layer is subscribed, and ask for a receipt or confirmation, as described in the example below. In the following exemplary scenario, <QueryEvent> logs the inquiry to the appropriate table in the database and asks to send a confirmation email back to the customer:

```
<QueryEvent>
    <Session_id>123af-5999c-b492</Session_id>
    <query type="log" table="customer-service">
        <field name="inquiry_text">
            I would like to get a quote on buying 34 units of your
            product at  corporate rate, Thanks. Jim
        </field>
        <field name="customer_id">1cb4af59</field>
    </query>
    <finally action="email">
        <inquiry-reponse/>
    </finally>
    <receipt/>
</QueryEvent>
```

The Data Layer Realm's Structured Query Language Service, SQLService, may then update the database table(s) and field(s) indicated in the <query> element. After the query is processed, components in the Data Layer may process the instructions in the <finally> element. In this example, the Data Layer sends a confirmation email to the customer using the template "inquiry-response."

While it may seem counterintuitive for the Data Layer Realm to handle email confirmations in the foregoing manner (as opposed to the Application Realm), those of skill in the art will appreciate that the Data Layer is designed to deal with external data sources; as such, the Data Layer Realm may support SOAP for inter-application interaction and interactions with external software packages such as Exchange Server and others.

Next, components in the Data Layer may return a receipt to the CustomerService's EventListener which originated the query as an indication of success or failure. The EventListener then may transfer control to the Formatter, which in turn may send the appropriate confirmation Message object, in XML or an internal data communication dialect, to the Server Gateway for external transmission. The Server Gateway may then redirect the data to the Soft-Spot Gateway shown in Fig. 2. In this embodiment, the SoftSpots may transform the internal communication markup, for example, into the appropriate client-side markup, which may be WAP/WML, HTML, XML, VoiceXML, and so forth. The functionality of the system is markup language independent.

THE HDA SERVER OPERATING SYSTEM

The HDA Server OS represents one embodiment of the infrastructure which provides operating system level Services to the HDA Server of the present invention. The HDA Server OS may manage typical low level operating system tasks, such as: resource management; application instantiation; application updates; booting and shutting down; low-level messaging; robustness or fault-tolerance, for example, dealing with system failure (overload, unexpected VM exit, hardware failure, network failure, and so forth); and security and authentication. The foregoing list is not intended to be exhaustive, and is provided by way of example only, and not by way of limitation.

Figure 7 is a simplified block diagram illustrating one embodiment of an HDA Server operating system for use in conjunction with an HDA Server. The HDA Server OS 700 resides in the OS Realm 210 illustrated in Fig. 2. The elements shown in OS Realm 210 are also shown in Fig. 7 as components of OS 700. Specifically, OS 700

may generally include a Migrator 701, a Packager 702, the AppInstaller 703 discussed above, a ServiceInstaller 704, a VMInstaller 705, a BootManager 706, a TopologyManager 707, a TopologyDescriptor 708, and VM Monitors 709. Examples of the general operation of each of these components are set forth below.

As noted above, OS 700 is a unique operating system. In particular, OS 700 is a high-level operating system designed specifically to be distributed; whereas typical operating systems are designed to operate within particular hardware constraints, the distributed architecture of OS 700 enables the OS to recognize available "hardware" as only a set of VMs. OS 700 may maintain a Topology list of all available and dormant VMs; the Topology list may be changed dynamically. Associated with each VM, OS 700 employs a VMMonitor 709 which monitors the VM's usage (load, memory, performance).

The VMMonitor

VMMonitor 709 may be in the form of a static Java class that may be "planted" by OS 700 at boot time (or any time a VM is spawned). VMMonitor 709 may advantageously use Java's Runtime information, but other embodiments are within the scope of the invention; for example, VMMonitor may employ the more advanced profiling available in version 1.2 of the Java Development Kit (JDK1.2). Importantly, VMMonitor 709 is a Transmitter, *i.e.* it may publish low-level events such as <memory-threshold-exceeded> or <load-critical>, which events may then be addressed by other components of OS 700. In effect, a VMMonitor 709 is a "dumb monitor," which means it never takes action.

Another important feature of VMMonitor 709 is the fact that it serves as an anchor (or a port) to the actual JavaVM it monitors. In one embodiment, it may be desirable that only one VMMonitor 709 be assigned per JavaVM.

Every VMMonitor 709 may have a unique ID which serves as identification with respect to the rest of the OS. This ID also serves as the entry for an individual VMMonitor 709 in the Registry, so that each VMMonitor 709 may be called by other objects. Any object having a reference to VMMonitor 709 may call it to request real-time information related to the JavaVM it represents. For example, an object may call getAvailableMemory() or getLoad(). Generally, a VMMonitor 709 may be non-migrative (Migration is discussed in detail below) and may not be serializable.

It should be noted here that, as used herein, the term VM may not necessarily be intended as a specific reference the JavaVM directly. That is, when reference is made to a VM, operation of the VMMonitor 709 is generally implicated.

TOPOLOGY

Topology objects may be used collectively to define a landscape of PMs and VMs on which the HDA Server OS can run. Topology management is crucial to the proper functioning of the HDA Server OS, since both Migration and Installation depend upon the current Topology. Topology is not a static concept; rather, in a running HDA Server OS, Topology changes constantly as new PMs and VMs are added or removed from the system architecture. As a consequence, the data managed by the TopologyManager 707 is mission-critical. A current snap-shot of the available Topology may preferably persist and be maintained in the LDAP server (discussed

above with reference to Fig. 3) so that the Topology may easily be reconstructed if the TopologyDescriptor 708 fails at any time during runtime.

As shown in Figs. 2 and 7, Topology data may be contained in TopologyDescriptor object 708. Figures 8A and 8B illustrate an XML representation of one embodiment of a TopologyDescriptor employed by an HDA Server OS. Though represented in XML form in Figs. 8A and 8B, TopologyDescriptor 708 may actually be maintained as a DOM document.

In Figs. 8A and 8B, the <vm> element actually refers to a VMMonitor object, such as VMMonitor 709, and not a JavaVM.

The TopologyDescriptor object 708 holds and maintains Topology data (as a DOM document, for example) during runtime. New data, such as a newly available resource (*i.e.* a new VM or host PM), for example, may be dynamically added through the Management Console (MC). The MC may notify TopologyManager 707 of the availability of the new resource, whereupon TopologyManager 707 may call the installer, for example VMInstaller 704, to install the resource and its VMMonitor 709. After the VMMonitor 709 is installed for each new resource, TopologyManager 707 may register each new resource with the Registry and add the entry describing the resource into TopologyDescriptor 708.

Upon notification of the Registry and update of TopologyDescriptor 708, the newly added resource (VM or PM) may be immediately available for use by the OS. Additionally, TopologyManager 707 may also maintain a list of dormant VMs, *i.e.* VMs which are not presently active but which may be spawned at runtime by request.

MIGRATION

In accordance with one particularly desirable aspect of the present HDA system, the OS may enable Migration and Adaptivity. Migration means that running application components may be selectively moved to a different address space (*i.e.* to a different PM or VM) at runtime. In fact, The entire HDA Server may move to new hardware resources while running.

By way of introduction, and referring back to Fig. 7, it is important to note that the Migrator object 701 is different from an Installer. For example, Migrator 701 may specifically handle the moving or migration of "living" runtime objects that are currently known and even presently in use by the HDA Server. Installers, on the other hand (such as AppInstaller 703, ServiceInstaller 704, and VMInstaller 705, for example) may only move un-initialized (*i.e.* presently unknown and unusable) application components.

In one embodiment, Migration may be achieved through rigorous Topology management and resource allocation. The HDA Server OS 700 manages Migration on the Topology assigned to it. Migration takes into account the instantaneous state of each VM at the moment the MigrationRequest object is published. As noted above, TopologyManager 707 may maintain a list of available and dormant VMs. This list may be dynamically sorted by resource consumption; in such an embodiment, for instance, the first VM on the list may be the least active, and consequently the least loaded VM in the Topology, whereas the last VM on the list may be the VM with the highest load, and consequently the fewest available resources.

In the foregoing embodiment, Migrator 701 may attempt to migrate application components first to those VMs identified as relatively "empty" or with very little load. Migrator 701 may also calculate in advance the implications of a prospective Migration. For example, Migrator 701 may not migrate an application component to a VM which is currently (pre-Migration) at a 55% load if the contemplated Migration may cause the VM to be at a 65% load (post-Migration) when the load threshold for that VM is only 60%.

If system traffic conditions are such that no existing VM is an acceptable Migration candidate for a particular load, Migrator 701 may request the OS 700 to spawn new VMs dynamically and to use the newly-spawned VMs for accommodating the migrated load.

Importantly, it is not necessary that all objects belonging to a particular application component be moved in order for the component as a whole to Migrate. In one embodiment, only serializable objects may move, and the ORB (*e.g.* Voyager) may oversee marshalling and transmitting all the objects referenced by the application component over the wire.

Migration may advantageously be managed in two ways during runtime: automatically, *i.e.* initiated and controlled completely by OS 700; and through the MC discussed above, *i.e.* with human intervention. Migration may generally occur at any given time, when desired. This is so because the HDA Server itself may never be apprised of the machines (PMs and VMs) on which it is running -- in this embodiment, the HDA Server OS 700 takes care of hiding those details from the HDA Server. Moving objects from one VM to another across the network is practically instantaneous;

the object "stub" at its original location may preferably stay active until the object clone at the new location is ready, at which time the original object is freed to be "garbage collected" and its resources become available to other system components.

Given the foregoing, those of skill in the art will appreciate that neither the ability to support Migration, nor the actual Migration itself, impose any timing constraints on the runtime. Rather than timing constraints, the HDA Server OS 700 acknowledges resource limitations. The VMMonitor 709 may evaluate any Migration request and automatically build an internal list of Migration targets. This list can only be built by elements of OS 700. In other words, an operator cannot decide the order in which components should or will Migrate (unless such an operator were to issue individual MigrationRequest objects for a single component at a time). This is so because OS 700 is the only entity which has runtime information about all the resources available for Migration, as well as load and memory information.

Once the list of potential Migration targets is established, a MigrationManager (the Service that manages Migrator objects) may begin executing the Migration in order, spawning new Migrator instances for each object to be moved. Again, Migration happens while a component or object is actually running; end-users will not be noticeably affected. The Migrator object may de-register the old object and register the new object in the Registry.

It will be appreciated by those of skill in the art that ORBs are suitable for moving objects. The following is an example of one embodiment of a process employed by an ORB for "moving" an object:

Object O1 runs on VM1. ORB uses VM1 to create an exact copy of O1's current state when the MigrationRequest is issued (this process is commonly called marshalling). ORB creates a new instance of O1 (O2) on VM2 and initializes it to the same state as O1. Once O2 is fully and successfully initialized, ORB frees O1 to be garbage collected (*i.e.* O1 dies). In this example, O2 is an exact copy of O1 on a different VM (which may be anywhere on the network). Subsequent to O2's initialization, all requests to O1 are immediately re-routed to O2.

Figure 9A is an example of one embodiment of a Migration Request event. Figure 9B is a simplified flow chart illustrating one embodiment of a Migration employed by an HDA Server OS. With respect to Fig. 9A, it will be appreciated that there may be several Migration targets in any given MigrationRequest event, as indicated by the ". . . etc." line. Further, the order in which each <target> is listed in the event document is not important; MigrationManager may make an internal decision based upon each target's priority and statistics, as well as the current Topology and system load conditions.

In the exemplary procedure illustrated in Fig. 9B, the <migration-request> event is issued within a MigrationRequest object (a subclass of Message) at block 901. The <migration-request> event may be routed by the Registry to the MigrationManager Service as shown at block 902; MigrationManager provides a new instance of a Migrator object (element 701 in Fig. 7) at block 903 to handle this <migration-request> (as noted above, Migrator 701 is an ORB agent, and is therefore, movable).

Next, as represented at block 904, MigrationManager calls the TopologyManager's getMigrateableResources() object. This call may return a

collection of VMMonitors (709 in Fig. 7), which may be sorted from least heavily loaded to most heavily loaded (this list may exclude those resources which are loaded beyond a predefined threshold). At decision block 905, MigrationManager may attempt to find the best VM for accommodating the Migration, *i.e.* the VM which has the lightest load and can support the target (judging from the VM's statistics).

As indicated by the "No" arrow from decision block 905, if none of the available VMs is an adequate candidate, MigrationManager may request TopologyManager (707 in Fig. 7) to spawn a new VM (block 906) from a list of Dormant VMs maintained by TopologyManager 707.

At block 907, MigrationManager may update the Migrator's DOM document with the <target-vm> element which may have been identified either in block 905 or 906. The <target-vm> element may also be added to the <migration-request> document shown in Fig. 9A.

After the resources to be tapped have been determined as set forth above, Migrator may move to the target object and optionally calls its prepareToMigrate() method, if the object has been provided with one (block 908). This may be characterized as an optional step, because even in the case where an object does not have a prepareToMigrate() method, the object may still Migrate.

If the target object is not serializable or not known to the Registry, however, Migration may not be possible; in this case, Migrator may return an error message to the MigrationManager, which in turn may notify the MC that Migration may fail.

At block 909, Migrator may employ an ORB so as to move the target object to the VM (or more precisely, the VMMonitor) identified in the <target-vm> element.

Block 910 represents a "clean up" step of sorts, where the Migrator may notify the Registry of a successful relocation of the application component, and the ORB may free the target object from its original location to be garbage collected. The Migrator dies upon successful completion of the Migration, as represented at block 911.

Packaging

The term "Packaging" in the HDA Server OS generally refers to the process of retaining data and object state so that both data and state may be retrieved and reconstructed in case of failure. With reference now to Figs. 3 and 7, the Packager module 702 may also be in communication with the LDAP server 349, although in a slightly different way than the Registry 340. The PackagingManager Service manages Packager objects which may be used by the HDA Server OS at critical moments. In effect, the PackagingManager may function very much like a specialized miniature Registry.

In operation, objects which request (or are required by the interfaces they implement, for example) to be packaged may register with PackagingManager and use that Service to retain their state in the LDAP server at critical moments. Objects are retained by being serialized (Java serialization, for example, is known in the art) into XML. These objects may subsequently be de-serialized easily from XML. De-Serialization may also managed by Packager 702. When recovery from failure is required, the HDA Server OS may iterate through all the serialized objects which need recovery and may then call Packager 702 to re-create each object with the state it had prior to the failure.

In one exemplary embodiment, a process of Packaging an object's state may proceed substantially as follows. The following represents one embodiment of packaging, with reference to the TopologyManager, by way of example only, and not by way of limitation. Initially, the TopologyManager may register at the PackagingManager Service to enable subsequent packaging when desired. After registration, the TopologyManager may send a <packaging-request> event to the PackagingManager, at which time the PackagingManager may provide a new instance of a Packager object and pass (to the Packager object's constructor) a reference to the TopologyManager. The Packager may then serialize the TopologyManager into XML and save the resulting XML to the LDAP server.

INSTALLERS

As noted above, the HDA Server OS allows the HDA Server to install new application components during runtime. Under certain conditions, it may also be desirable to remove application components during runtime. Install and un-install processes may be managed by the HDA Server OS using Installer objects. The InstallerManager Service may maintain and launch the various Installers, such as elements 703-705 in Fig. 7, for example. All Installer objects may be ORB agents, *i.e.* they may be moveable as discussed above. Installer objects generally function in a similar fashion as Migrator objects detailed above, however, Installers work with application components which are not yet part of the HDA Server runtime.

In one advantageous embodiment, all concrete Installers (*i.e.* not abstract) may be characterized as PluginInstallers. In such an embodiment, only Plugin objects may be installed, each type of Installer may be considered a specialized case of

PluginInstaller, and all Applications or application components may be considered a type of Plugin. While Applications, Plugins, application components, and Services may all be installed during runtime, Realms may not.

In other words, AppInstaller 703, ServiceInstaller 704, and VMInstaller 705 may be considered as part of the base class (PluginInstaller), and the components installed may be considered Plugins. Recalling the installation of a Plugin discussed above, the AppInstaller may transfer control of the installation to the Application Realm, which then delegates the process to PluginManager.

PluginManager may attach the Plugin to the specified <parent> element in the specified Service type (for example, a SessionAware Service: WAManager), and call the HDA Server OS's TopologyManager to find the best or most suitable VM for installation in the Application Realm. PluginManager may load and initialize the Plugin. Finally, the Plugin may finish initialization by loading and initializing its own Formatter and Transmitter.

Applications may be installed using the AppInstaller installer (which may itself be a type of PluginInstaller). That is, AppInstaller may typically deal only with Applications in the Application Realm, whereas the base class (PluginInstaller) may deal directly with installing the application components which are not specifically self-contained Applications. Additionally, the AppInstaller may install components used by larger Applications or system-level applications (*e.g.* data access Services).

As noted above, any Plugin (*i.e.* Application, application component, VM, Service, and so forth) has the potential to be installed and removed by Installers during runtime. The "location" at which a given Plugin is to be installed is defined in the

Plugin's descriptor file. The actual success of installation or removal of the Plugin may ultimately depend upon runtime vetoing power exercised by the component to which the Plugin is to be attached. For example, a Service may reject a Plugin installation request.

The process for installing an application component may be almost identical to the Application installation process as set forth in detail above with reference to Plugins. To distinguish the type of installation at the system level, however, a minor difference may be provided in the Plugin descriptor for an application component, *i.e.* its <plugin-type> may be defined as "application component" and it may further include a <parent> component, which represents the Application to which the application component belongs.

In the case of application component installation, PluginManager may call registerApplicationComponent(Plugin) (a method common to all application-level components in the HDA system) on the <parent> component. The <parent> component may then add the new Plugin to its list of application components, and follow any instructions provided in the <parent> element.

For example, a <sub-process> instruction may be provided in the plugin descriptor, which may refer to the type of Services for which the parent may use the new Plugin. As an example, a request which begins with the string "pag" may be understood to be a "pager" communication event, for example, and may be interpreted by the parent as delegable to the newly installed Plugin. Another instruction, for example <propagate-events>, may mean that the new Plugin will subscribe to events through the parent component, and not independently. It will be appreciated that many

different types of instructions may be provided in the <parent> component so as to govern interaction between the parent and the new component in a desired manner.

The parent component may then call PluginManager's updateRegistry(Plugin, String, String[]), PluginManager may notify the Registry and pass along all the data provided by the Plugin. Upon successful registration (this is a Transactional process, as discussed above), the Plugin's internal state is set to running, at which time to is open to receive incoming events.

New Services may also be installed during runtime. As noted briefly above, in one embodiment Services may be considered Plugins, and therefore the installation process may be very similar to that detailed above. Services are very simple Plugins in terms of installation. The ServiceInstaller (704 in Fig. 7) only targets the Realm into which the Service will be installed and updates the Registry. Since a new Service does not hold any Applications or application components, installation may be a very simple and straightforward process.

To distinguish the type of installation at the system level, a minor difference may be provided in the Plugin descriptor for a Service, *i.e.* its <plugin-type> may be defined as "service" and it may further include a <ServiceInstaller> element, which may include a declaration attribute only applicable to Services.

In the case of a Service installation, for example, PluginManager may be configured accordingly (PluginManager knows a Service is to be installed, since control of the installation was passed by the ServiceInstaller). PluginManager may then call registerService(Plugin) (a method which may be common to all Realms) on the <realm> component of the Plugin descriptor. The Realm may then add the service to its list of

available Services. The Realm may then read the "declare" attribute of the <ServiceInstaller> element discussed above. The declare attribute refers to Realm-level declarations and relates to defining or "declaring" the Service to be a SessionAware Service or a Daemon Service.

The Realm may next call PluginManager's updateRegistry(Plugin, String, String[]) and notify the Registry, passing along all the data provided by the Plugin. Upon successful registration (a Transactional process), the new Plugin's internal state is set to running.

The loosely coupled architecture of the HDA Server and OS allow for efficient resource management and runtime stability through effective implementation of VMs. To accommodate rapid adaptation to dynamically changing system load conditions, it may frequently be desirable to install or to remove new VMs at runtime. Installation in this context may be different from the Plugin installation described above.

It should be noted again that the HDA Server OS recognizes VMs only as VMMonitors (depicted as element 709 in Fig. 7). VMMonitors may be considered Plugins; in fact, VMMonitors are very simple Plugins in terms of installation. Similar to a ServiceInstaller 704, the VMInstaller 705 only targets the Realm into which the VMMonitor (and its JavaVM) will be installed. Since VMs are not registered (*i.e.* they are "dumb" monitors), the Registry need not be updated with each installation or removal; further, no involvement of PluginManager is required in the VMMonitor installation process. An additional requirement, however, involves the HDA Server OS making a system call which spawns a new VM process (*i.e.* in the form of "java VMInstaller"). The process ends with the updating of the TopologyDescriptor.

The Plugin descriptor for a VMMonitor may be provided with a <plugin-type> of "vm." The runtime installation may generally be initiated by TopologyManager (at boot time, the installation may be initialized by BootManager, discussed below). VMInstaller may spawn a new JavaVM by calling "java VMMonitor" -- this is a system call. VMInstaller may then load and initialize the VMMonitor and call the HDA Server OS to update the TopologyManager on the newly available VM.

BOOTING THE SYSTEM

Booting the HDA Server system is a complex process; boot-up generally may be characterized as having two distinct phases. First, booting requires setting up the HDA Server OS on a set of PMs and VMs and readying the HDA Server OS for the applications which run on it. The HDA Server OS spreads its Realms across the resources available to it and hooks into the Management Console discussed above. As far as the HDA Server OS is concerned, this is the end of the boot process, even though Services, Applications, and other Plugins may not yet be initialized. The second phase of booting involves the Application Layer, *i.e.* installing the various Plugins (Services, Applications, and application components). The end of the second phase of the boot process occurs when the Server Gateway and one or more optional gateways are attached to the HDA Server.

The process of booting the HDA system may generally be managed by a BootManager object. BootManager may be a Java application which preferably exits as soon as it has finished booting the HDA Server OS.

Figure 10 illustrates one embodiment of an svcs.xml file which may govern the boot process for an HDA system. Figure 11 is a simplified flow chart illustrating one

embodiment of an HDA system boot process. The following description, with reference to Figs. 10 and 11, is provided by way of example only, and not by way of limitation. In preparation for booting, an administrator or other operator may first prepare a running ORB server on each PM which will be available to the HDA system. In one embodiment, booting may begin in the normal Java command-line fashion, for example:

java BootManager -topology TopologyDesc.xml -bootsvcs svcs.xml

As indicated at block 1101 in Fig. 11, BootManager may first obtain and follow the instructions provided in the svcs.xml file (one example of which is shown in Fig. 10). The svcs.xml file defines the system-level Services which need to be initialized external to the Realms, for example the Registry, the MessageFactory, the TransactionFactory, and the Server Gateway. The initialization of these Services by the BootManager is represented at block 1102.

Next, at block 1103, BootManager may initialize a TopologyManager and build a TopologyDescriptor from the TopologyDesc.xml file (or any file that was passed to it as its "topology" command-line argument). An example of a runtime TopologyDescriptor is shown in Figs. 8A and 8B; in a boot-time TopologyDescriptor, IDs may not have been assigned to the VMs in the case where the XML file does not contain VM IDs.

TopologyManager may then initialize the various OS Realm Services, *i.e.* Migrator, Installers, and Packager, and register these Services and itself with the Registry, as indicated at block 1104. Upon completion of this task by the TopologyManager, BootManager may publish a <boot> event to the Registry at block 1105.

Upon receipt of the <boot> event, TopologyManager may begin deploying VMInstallers according to the addresses in the TopologyDescriptor. Once the VMs have been installed successfully, the TopologyManager may assign Realms to the VMs and register the Realms with the Registry (blocks 1106 and 1107).

TopologyManager publishes <boot-one-complete> event (block 1108). At this moment, as indicated at block 1109, phase one of the boot process is complete.

Phase two of the boot process may begin when the BootManager receives the <boot-one-complete> event and verifies state (block 1121). When the system state is verified, BootManager may publish an <initialize-Realms> event; all Realms may preferably be subscribed to this event by default. Next, Realms begin initialization. These processes are represented at block 1122.

Further, BootManager may launch all the Plugins and Applications (block 1123). In one embodiment, software is launched in the following order: Applications; other Plugins; and Resources. Finally, all Plugins and Applications which require registration take action to register themselves in the Registry, as indicated at block 1124. At this moment, phase two of the boot process is complete (block 1125).

SERVER GATEWAY

Figure 12 is a simplified block diagram illustrating one embodiment of a Server Gateway which may be employed by an HDA Server. In the Fig. 12 embodiment, HDA Server 200, Server Gateway 270, and Registry 240 correspond to the elements shown in Fig. 2 having like reference numerals. Similarly, an exemplary interaction between Gateway Server 270, Session 571, and MessageFactory 551 was set forth above with

reference to Fig. 3. It should be noted that SoftSpot Gateway (element 280 in Fig. 2) has been omitted from Fig. 12 for clarity.

In one embodiment of HDA Server 200, the Server Gateway 270 may contain the logic for managing Sessions 571 for the entire HDA Server 200. When Server Gateway 270 accepts a login request from a SoftSpot 281 (or any other client conforming to the appropriate protocol), for example, a SessionManager at Server Gateway 270 may control the Session by allocating a Session object 571 to that connection. The Session 571 is the interface between SoftSpot 281 and HDA Server 200. From the SoftSpot's perspective, the Session object 571 is the HDA Server 200; conversely, from the HDA system's perspective, the Session object 571 is the SoftSpot 281.

Two concrete classes may be provided for implementation of the Session functionality: the Session class; and the SecureSession class, which may provide additional infrastructure for validation of requests from a client such as SoftSpot 281.

In operation, Session 571 in HDA Server 200 may be managed at the Message level. One advantage to this methodology is that, as the various components of HDA Server 200 are negotiating a query or a Transaction, each component may always pass the associated Session ID simultaneously with Message data so that all the data belonging to a specific client Session will remain constant. A data Message having the appropriate Session ID appended thereto is denoted as element 572 in Fig. 12.

As noted briefly above, SoftSpot 281 may be a Servlet responsible for pumping events through Server Gateway 270 (in the form of Sessions 571) to HDA Server 200. As HDA Server 200 generates responses to those events, the responses go back to

Session, 571, and subsequently to the appropriate SoftSpot 281. The HDA Server OS may be uninvolved in these activities until Session 571 shuts down or is otherwise terminated, at which point a SessionManager may execute clean-up work and re-allocate system resources accordingly.

In one embodiment, Server Gateway 270, through Session 571, wraps incoming events (which, as noted above, may be in XML or some other markup language) into a Message object 572 which contains a DOM representation of the original message data. Server Gateway 270 may then append additional information such as, for example, SessionID, a client identifier (clientID), a seller identifier (sellerID), and so forth. Server Gateway 270 may also subscribe to the return event of Message 572. In another embodiment, Server Gateway 270 may be incorporated in an Enterprise Java Bean (EJB)-compliant architecture, for example.

Server Gateway 270 typically receives requests from the SoftSpots 281. By way of introduction: SoftSpots 281 may be built around the Java Servlet API, for example. SoftSpots 281 may serve as "proxies" which may adapt the User Interface (UI) markup which is sent to the client's User Agent in real-time. For example, SoftSpots 281 may adapt or convert the "neutral" markup (SOAP, for instance) produced through HDA Server 200 to WAP/WML, VoiceXML, HTML, or any other XML dialect.

Figure 13 is a simplified flow chart illustrating the operation of one embodiment of a Server Gateway employed in conjunction with an HDA Server. In one embodiment, a SoftSpot holds a remote reference to the Server Gateway through an ORB.

To initiate a new Session, SoftSpot asks the Server Gateway to create a new Session for this client (for example, by remotely calling the createNewSession() on Server Gateway).

SoftSpot sends a request in an XML String to the Server Gateway, as indicated at block 1301 in Fig. 13. If this is a new Session, Server Gateway may spawn a new instance of a Session (from the pool); in every case (whether the Session is new or not), Server Gateway may relay the XML String to the Session responsible for this SoftSpot (block 1302).

Upon receipt of the XML, Session may call the static createMessage(String) on MessageFactory, which may then create a DOM representation of the XML String as a Message object (block 1303). MessageFactory preferably adds a list of target "subscribers" to the Message object. As indicated at block 1304, MessageFactory may stamp a MessageID to the Message, so that the Message may be identified; additionally, Session may subscribe to the <response> event with its SessionID and the specific MessageID for this Message.

At block 1305, Session may then multicast the Message to all subscribers. As block 1306 indicates, Session may receive the <response> Message from the HDA Server and transmit that Message to the SoftSpot which issued the request.

DATABASE ACCESS

Database access may be the exclusive responsibility of the Data Layer Realm and may generally be accomplished through a set of Plugins which may be modified, installed, or removed during runtime as discussed above; this runtime alteration is important for adding new data manipulation Services to the HDA Server. Querying a

database may be done by sending a Message object with a request to which the Data Layer is subscribed, for example (by default: a <data> Message).

Figure 14 is a simplified block diagram illustrating the operation of one embodiment of database access which may be employed by an HDA Server. One or more database access Plugins, such as Structured Query Language (SQL) Plugin 1401 and Common Object Request Broker Architecture (CORBA) Plugin 1402, may be installed in the Data Layer Realm. It will be appreciated by those of skill in the art that other Plugins may perform similarly; only two are illustrated in Fig. 14 for clarity. The Plugin 1401,1402 may decide to which Business object (database interface) to transmit the Message object which contains a request for data.

The Business object has the actual "knowledge" of the data source (*i.e.* it is the Business object which is capable of JDBC data transfer). The Plugin 1401, 1402 may work against a remote object 1403, which serves as a front end to the database 1404. The data source, *i.e.* database 1404, may be an SQL server, an object database, or an ORB which serves as a front end to data on a mainframe computer, for example. Operation of the present invention is not limited by the nature or architecture of database 1404.

When a Message arrives at a database access Plugin 1401,1402, the Plugin may determine the type of database request and may subsequently invoke the appropriate method on the front end to the database 1404. This may typically be accomplished through a "translation" of the incoming <data> element's markup into the appropriate query language.

For example, if the database 1404 is SQL, the <data> will be embedded into an SQL query. In order to be more efficient, a <data> query may "force" a query statement by directly specifying the query string in a <force-query> element. This may advantageously spare the translation phase.

In one embodiment, the Plugin 1401,1402 and the front end 1403 may be on different machines; in such a situation, the interaction between these two elements may be effectuated through ORB remote method calls.

Figure 15 is a simplified block diagram illustrating the operation of another embodiment of database access which may be employed by an HDA Server. In the Fig. 15 embodiment, an implementation of database access for Open Data Base Connectivity (ODBC) is shown.

The front end 1503, generally corresponding to front end 1403, may serve two functions in particular: ensuring that the SQL is properly formatted for communication with data server 1505; and pooling connections. With respect to pooling, those of skill in the art will recognize that connections to the database 1504 are expensive (in terms of system resources) to create. In this regard, connection pooling may be implemented to save system overhead and to expedite the query process. When the Data Layer is finished with a connection, the connection may not be terminated, but rather may be saved in a pool of inactive connections. This inactive connection may be reused (without having to be recreated) as system load requires.

LOAD BALANCING

During operation, the system may be under heavy load such that physical or virtual resources become scarce or unavailable to entities. By way of example, system

resources which may be relevant to operation of the HDA Server include at least the following: memory; CPU time; and threads.

At least two types of load balancing methodologies may be considered by the system: optimistic; and pessimistic. In accordance with optimistic load balancing techniques, the initial deployment of the system may take advantage of available hardware by deploying a system which is capable of handling any reasonable amount of load. In this context, "reasonable" may defined by the enterprise, based upon projected usage. In accordance with pessimistic load balancing techniques, the original assumptions may be determined to have been incorrect, *i.e.* the system did not properly take into account site demand. A surge of demand may exceed the constraints set for the optimistic load balancing.

In other words, optimistic load balancing generally occurs as the OS attempts to determine the best location to create Sessions and Plugins; this may typically be done at boot and when initially creating an instance of a Plugin. Pessimistic load balancing, on the other hand, may involve duplicating Plugins (*i.e.* creating new instances), and may be accomplished dynamically during runtime.

An underlying assumption of optimistic load balancing is that, for the lifetime of a user Session on a given site, an upper threshold of required system resources may be determined or approximated. For a typical Session, for example, it may be assumed that the projected load, based for instance upon the number of Sessions, may be estimated. Given such an assumption, system load may be measured based upon the current number of Sessions. Following are two examples of proactive measures which may be

taken to balance system load; these measures are provided by way of example, and are not intended to be representative of an exhaustive list.

Since Plugins actually perform computations which require memory and processor time, it may be desirable to distribute multiple instances of the same Plugin across more than one VM and PM, for example. In one embodiment, the system may duplicate Plugins by creating new instances on other machines; this enables the HDA Server to spread events between and among the several instances of the same Plugin in accordance with system demand and the load at each respective VM. Consequently, multiple requests to a particular Plugin may be advantageously distributed around the system.

As detailed above, the Session object is the interface between the SoftSpot and the HDA Server. In terms of overall system resources, it may be beneficial to limit the number of Sessions running on the same machine. The decision concerning Session location may be made when each particular Session is created. That is, in one embodiment, when a new Session is created, the Session Manager may determine which VM has the lightest load and assign the Session object to that VM.

The same general approach may be applied with respect to installing Plugins; the HDA Server OS may exercise discretion, whether at boot or during runtime, concerning where to install the Plugin. The PluginInstaller engineer may also determine which VM has the lightest load at a given instant and install the Plugin on that VM.

It will be appreciated, however, that Plugin resource consumption (*i.e.* cost) may vary considerably depending upon the Plugin type and demand. While it is generally desirable to install more expensive Plugins on more powerful CPUs, determining or

accurately projecting the cost of a particular Plugin during creation (installation) may be complicated; such a determination may, itself, present a substantial cost in terms of system resources. As a result, a Plugin may initially be installed in a location which later proves to be less than optimal. In this regard, the migrative aspect of the HDA Server and OS of the present invention provides an efficient solution to the load balancing problem. That is, if a very expensive Plugin is installed in an inappropriate location, the HDA system may migrate that Plugin to a more acceptable location.

As noted briefly above, pessimistic load balancing methodologies assist the system through creating new Plugin instances; an important consideration may be to determine how to route Message objects to the new instances of the Plugin.

When the system decides, through evaluation of load conditions, to create a new instance of a Plugin, the best VM candidate to receive the new instance must be identified. In one embodiment, for example, the system may execute the following procedure to identify the best VM.

First, the system must identify the correct space, *e.g.* the Realm to which the Plugin must be assigned. Next, the system may iterate across all the VMs in the Realm to locate the VM with the lightest load. If the identified VM has not passed an upper load threshold, and installing the new Plugin instance will not bring the load over the threshold, the VM may be used for the new instance. If the VM is over the threshold, the system may attempt to create a new VM, for example, by looking up the adhoc VMs section of the boot.xml file.

The adhoc VMs section may advantageously list where and with what settings a VM may be created in an emergency. This section of the boot file may be determined

or altered by a system administrator, for example, and may be based upon how extra hardware is to be allocated to the system when necessary.

If the Realm has sufficient capacity for another VM, the system may create the VM and delete the corresponding portion of the adhoc VMs list; subsequently, the system may allocate the next instance of the Plugin to the newly created VM. If every VM is over its respective assigned load threshold, and system resources are inadequate to create a new VM, the pessimistic load balancing procedure may fail. In such a case, the system may investigate alternative resource allocation schemes, for example, migrating applications and objects in order to optimize overall system resource allocation.

It will be appreciated that certain restraints may be beneficial with respect to the foregoing procedure. For example, it may be preferable that certain Plugins not have more than one instance; similarly, it may be preferable that a particular Plugin run on a particular VM. In such circumstances, a system administrator may be able to override the foregoing load balancing mechanisms for such Plugins. In one embodiment, for example, a Plugin.xml file may be amended to include instructions limiting the number of allowable instances of a particular Plugin or specifying a VM required or preferred for one or more instances.

TagChangeHandler

The HDA Server and OS may employ a TagChangeHandler for handling changes in the Tags for one or more particular multiplexers (MUXs). With every change in a Plugin, *i.e.* a new instance or a Migration, for example, the MUX has to

determine if its Tags have changed. If the Tags have changed, the MUX may pass that information to the TagChangeHandler, which may respond in two possible ways.

In one embodiment, the TagChangeHandler may notify a Transmitter object about the changes; the Transmitter, then, may pass Tag change information on to the other elements of the system through publication. In another implementation, the TagChangeHandler may simply store the Tag changes internally; in such an embodiment, for example, the TagChangeHandler may apprise other elements of the system of changes upon request.

ServiceResourceProvider

Service Level Resources are resources available to Plugins and MUXs. Examples of these resources include an internal timer, an error handler, and so forth. Service Level Resources may be provided by invoking a ServiceResourceProvider interface with the name of the resource desired; ServiceResourceProvider may then return an instance of that Resource. For example, in the case where a Plugin wants to create a Message object, the Plugin may call the ServiceResourceProvider for the MessageFactory; as a result, the Plugin may receive a reference to the MessageFactory.

DefaultRequestMultiplexer

In accordance with still another embodiment, the HDA Server and OS may employ a DefaultRequestMultiplexer to provide the general infrastructure for managing Tags to all its subclasses. In effect, a subclass may query, "My child has changed its Tags, what does that mean to me?"

By way of example, Tag management may be accomplished in a "Magic Box." The Magic Box: knows the MUX's current Tag set; may be apprised of the change in

the child's Tags; may determine how the change affects the MUX's Tag set; and additionally, may report the actual changes in the MUX's Tag set (typically to the TagChangeHandler).

In the foregoing example, Magic Box may return an object (called, for example, TagDeltas), which may contain two collections: the Tags which have been added; and the Tags which have been removed. It will be appreciated that both collections may be null (indicating no changes).

The DefaultCoreMultiplexer may employ an EventPostingTagChangeHandler implementation, *i.e.* the embodiment in which the TagChangeHandler publishes Tag set changes to the system. The RequestMultiplexer may include setParent, getParent, and getChildren methods, *i.e.* the mechanisms to assign a MUX a parent as well as to query concerning a MUX's parent. In accordance with these methods, the MUX tree may be traversed in both directions. The addRequestHandler method may create a RequestHandler object.

In one embodiment, the RequestMultiplexer may assess whether the child is a RequestMultiplexer (the child could be a Plugin, for example); if the child is a RequestMultiplexer, the parent may invoke the setParent method to inform the child of its parent. The parent may then call the Magic Box, apprise it of the added RequestHandler, and request the "deltas" or changes. The Magic Box may then return the TagDeltas object (tags added and removed).

Subsequently, the RequestMultiplexer may invoke a handleChanges method to evaluate and to deal with the changes forwarded from the Magic Box. If there has been

a change in the Tag set, the TagChangeHandler's postNewTagChangeEvent method may invoke EventPostingTagChangeHandler to multicast the change to the system.

It will be appreciated that it is desirable to provide a controlled way to propagate changes in a Plugin's Tag set. When there is a change: the engineer does its activity; goes to the StorageTagHandler and asks what the change was; triggers the propagation by notifying the parent MUX's TagChangeHandler; and the parent calls its getTagChangesInChild method to get the child's TagDeltas object.

SECURITY

As noted above, a SoftSpot or other client may request the SessionManager to create a Session upon login. If requested by the client or required by the HDA Server, for example, the SessionManager may create a SecureSession, which may implement a security mechanism. In one embodiment, a SecureSession may generally be created and managed as follows.

A SecureSession may generate a special Message object requesting profile data, for example, user ID and password, which may be supplied by the SoftSpot which requested the SecureSession. A Plugin may receive the Message object and check the profile data in the database. The Plugin may then reply to the SecureSession with the user profile.

In one embodiment, the profile data may be stored internally by the SecureSession and appended to every subsequent Message object created by the SecureSession. Recipients of these Messages may decide, based upon the profile data, whether to process them.

In general, enterprise-level security may be managed in a fashion similar to Session-level security. Every HDA Server or OS entity concerned with security may be provided with a set of access permissions for user, group, and others. Read permission may enable access to entity properties or contents. Write permission may enable access to update controls or to enable/disable functionality. Execute permission may enable invocation of an executable script or program.

When an enterprise user or administrator wishes to access the HDA Server, for example, via the MC, a Secure Session may be created for that user. The Secure Session retrieves the appropriate profile data for that user from the database. As with Secure Session management, profile data may be appended to every Message object during the Secure Session.

In one embodiment, for example, a system administrator may set the permissions for all HDA Server and OS entities; permission data may be stored in the database. When an entity is created (a Plugin, for example), the entity may be informed of its own permissions, or set those permissions itself.

EXEMPLARY IMPLEMENTATION

Presently, the technological infrastructure for providing BSPs with a platform through which they can provide value-added services to their customers is inadequate to serve the needs of BSPs wishing to generate new revenue streams from offering new value-added services to their business customers..

In considering whether to create a Web presence, many small businesses come to the realization that creating and maintaining a stand-alone Web site does not make good business sense for their respective organizations. First, it may be difficult for

existing or prospective customers to find the new Web site; next, small businesses typically do not have a large inventory, and catalogue-like representation of products on a computer display may not present the products offered in the best light; and finally, Web sites, even when hosted externally, are difficult, time consuming, and potentially expensive to maintain and to manage.

Many current BSPs such as Application Service Providers (ASPs), CLECs, telecommunications service providers, or Internet Service Providers (ISPs), offer free Web site building and hosting, free "shopping cart" functionality, and free transaction processing for their customers. Such an approach to offering services ignores a significant problem inherent in the underlying technology by which the services are offered. In particular, the current e-commerce infrastructure, which relies upon the notion of catalogue-like "Web sites" and anonymous, impersonal customer relations, was designed solely with large retailers in mind; this conventional platform does not serve small businesses which require personalized, human interface with their customers.

It is expected that small businesses will increasingly associate themselves with, and aggregate in, BSP offerings. Therefore, the SoftSpot technology described herein is designed not only to be easily embedded in a Web site (should the seller already have one), but also to be useful to those sellers for whom having a Web site is not a viable option. All types of sellers may use WAP, PQA, or Voice-driven SoftSpots.

By way of example, many restaurants conclude that having a Web site does not make sense from a logistical or economic perspective. A very graceful solution may be to provide the restaurant with a SoftSpot through which a customer can interact (via a

wireless device or a regular telephone, for example) with the restaurant's host; advantageously, such interactions may be linked and integrated into the restaurant's data systems immediately. This is an appealing and economical solution since it requires no installation of new hardware or software.

The SoftSpot architecture creates a portable client-agnostic user-interface. Specifically, SoftSpots are user interfaces which adapt, in real-time, to the user-agent requesting them. For example, the same SoftSpot may be served to a WAP-enabled device (such as an Internet-enabled wireless telephone, for example), to a "standard," HTML-based Web browser, or to any voice user-agent (such as a telephone).

The SoftSpot architecture is not merely a request-time adaptation of content; the architecture contemplates providing services which are accessible not only from any networked device, but also from any number of different "domains." For example, a user employing a particular device to access the system from a specific domain may receive a domain-specific SoftSpot specifically formatted to the protocol required by the device. Similarly, another user accessing the system from a different domain may receive a different domain-specific SoftSpot. Moreover, SoftSpot Servlets are clients to a Marketplace Management System (MMS) maintained in the form of Services on the HDA Server described above. Consequently, the SoftSpots must interface with the Server Gateway and request and accept XML data in a desired markup language.

The SoftSpot architecture may be readily adapted to integrate seamlessly with the following technologies: adaptive XML-based content delivery; support for every type of networked UI; HTML (XHTML); WAP/WML; Applets; Shockwave/Flash; VXML; and PQA (Palm Query Application). The foregoing list is presented by way of

example only, and is not intended to be exhaustive; it will be understood from the following detailed description that the SoftSpot architecture may easily be extended to support future UI types and communication protocols.

Additionally, the system architecture of the present invention provides fast and reliable content delivery, secure and private communications and domain-specific "look & feel" characteristics. An e-commerce or other network-based operation employing SoftSpots may easily be configured by the seller, easily personalized for end users, and may offer persistent and stateful communications.

The HDA system described above may be configured to host and to manage a powerful, adaptive Customer Relationship Management (aCRM) application based upon SoftSpot technology.

In one embodiment, small business aggregators may use the HDA Server and OS system as a web-deployed ASP service to offer their sellers a rich and full set of online aCRM tools. From the foregoing, those of skill in the art will appreciate that the HDA Server may easily be configured to provide Services which may be embedded (transparently, for example) in aggregators' sites; the HDA Server's flexible, loosely coupled architecture and adaptive, migrative load management capabilities enable management of any number of such sites (or domains).

The HDA system may interface with the aggregators' various servers, applying "adaptation rules" to events that flow through the servers. Adaptation rules may be employed to configure seller data to Domain Rules, Business Rules, and Seller Rules. Use of the HDA system may be Session-based as described above, and the system may process data dynamically for every request.

By way of example: A request may arrive from a seller who sells on a particular domain; upon receiving the request, the system may retrieve requested data from a database (as noted above, data in the system may be maintained in XML or DOM form); the system may then apply Domain Rules to the data XML (if needed, as required by the domain), such as language, local currency, and other domain-specific formats, etc.; the system may additionally apply Business Rules to the data XML (if needed, as required by the type of transaction), such as pricing, profiling, etc.; finally, the system may apply Seller Rules to the data XML (if needed, as requested by the particular seller), such as alert preferences, tool set, etc.

As described above, the client of this data may be a SoftSpot, an Adaptive User Interface (AUI) application, which actually formats the XML data received from the HDA system to suit the requesting device, with the correct User Interface.

As detailed above, other important characteristics of the HDA system are:

Virtually infinite scalability -- meaning that more hardware and software resources may selectively be added, changed, or removed without taking the system down and without having any adverse impact on runtime system performance.

No single point of failure -- the HDA system design is such that no single component in the system can fail in a way which can bring the system down. Every critical component may be instantly replicated in case of failure, as discussed above.

Adaptable -- new aCRM applications may be deployed without bringing the system down and without adversely affecting runtime performance.

Flexible – the system may be configured easily to integrate with external, third-party applications through a flexible Server Gateway during runtime without bringing the system down and without adversely affecting runtime performance.

Adaptive CRM

In one embodiment, an adaptive Customer Relationship Management system (aCRM) is provided; the aCRM is generally constituted by elements deployed in each of the three Realms discussed in detail above.

The "Application Layer" (in the Application Realm) may hold the actual Business Logic for the various aCRM Applications. As noted above, Applications may be pluggable components, and may advantageously be written according to a standard HDA deployment protocol. Applications may interact with other Applications to discover each other's Services and to subscribe and/or publish data among themselves. Applications may be automatically deployable; that is, adding, modifying, or removing an Application may be done during runtime through the Management Console (MC) as set forth above.

The "OS Layer" (in the OS Realm) is preferably a highly distributed server-side operating system such as the HDA Server OS detailed above. The OS may provide low-level system Services such as, for example, process management, memory management, resource management, load balancing, replication (spawning new Plugin instances), and the like. In addition, the OS may provide innovative Services such as Migration, Topology Management and VM Management as discussed above.

Data Layer (in the Data Layer Realm) may interface with any number of data sources and supply data in XML or DOM form to the HDA Server. The Data Layer

may also interface with various types of data sources such as relational databases, object-oriented databases, and the like, and additionally or alternatively use CORBA to interface with legacy systems. The HDA Server's Data Layer may also support SOAP -- an upcoming industry standard for inter-application communication.

By virtue of the HDA system upon which the aCRM is structured, an aCRM Application has the following important features: it may be installed, modified, or removed seamlessly; it may be infinitely scalable to support hundreds of aggregators and thousands of end-users; its User Interface may be adapted in real time to wireless, voice, or HTML devices; aggregators and end-users may decide which set of Applications to use; and data may be utilized from many data sources of various types simultaneously.

In one embodiment, SoftSpots may have the following attributes, identified in accordance with two phases: presentation; and communication.

In the presentation phase, a SoftSpot may be expected to provide initial and concise information to a customer. In an e-commerce context, for example, SoftSpots may generally represent one item type -- not a quantity. For example, one SoftSpot may represent a set of items, such as a set of three books. In this e-commerce embodiment, SoftSpots may contain at least the seller's name and/or alias, a short item description, and the communication type(s) employed by the seller. Additionally, SoftSpots may also contain information concerning the following: seller rating; deal-type; time remaining until a deal closes or an offer expires (for example, during a timed auction); media files such as image, video, audio, and the like; and so forth.

In the communication phase, the SoftSpot may be expected to handle either a sale, a request for more information, or a request for customer service, for example. In other words, SoftSpots may open a communication channel between buyers and sellers; the SoftSpot communications channel may be real-time (as in chat) or deferred (as in email), for example.

In one exemplary embodiment, SoftSpots may support flexible, secure and private transactions in a wide variety of ways. The SoftSpot communications channel may be tied into the sellers' respective PO (Purchase Order) consoles. As an example, a seller may build a PO in real-time and send it to the buyer for approval.

The SoftSpot communications channel may be conveniently tied into a seller's inventory management system; in this embodiment, for example, inventory information may be retrieved from the inventory management system in addition to, or as an alternative to, retrieving the information directly from the seller.

The SoftSpot communications channel may additionally be tied into the sellers' respective customer service systems, for example, a software Customer Service Console (CSC) application; in this embodiment, information may be retrieved from a Frequently Asked Questions (FAQ) database, for example, through an interface with the CSC.

To support optimum flexibility, the HDA system of the present invention may allow a seller to issue any number of SoftSpots. SoftSpot "issuance" may be done automatically by the system, and additionally may be directly tied to the seller's console. Sellers may set preferences to control SoftSpot behavior as well as the "look & feel" of the UI. For example, a specific backdrop or frame arrangement may be provided for applicable Graphical User Interfaces (GUIs). In some instances, it may be

desirable to restrict a SoftSpot's access to certain communication types (such as telephone or real-time chat), whereas, on the other hand, it may be desirable to attach a pager or a cellular telephone alert to a SoftSpot event. Additionally, the system may be adapted to support logging of any or all types of SoftSpot activity.

As noted above, SoftSpots are highly focused AUI components; the operability of SoftSpots may be selectively limited to two primary functions: to serve as a networked customer interface for the seller; and to allow maximum flexibility for customer acquisition, satisfaction, and retention.

By way of further background, SoftSpots may not create (what are commonly known as) Web pages; that is, SoftSpots may only interact with and serve as gateways to a common communications channel. For example, the email or chat applications mentioned briefly above may not be part of the SoftSpot itself, but may be operable through the SoftSpot.

In that regard, it should be appreciated that SoftSpots only interact with, and are gateways to, the MMS (the MMS may comprise a transaction and payment recordation and reporting system) maintained on the HDA system. That is, the SoftSpots themselves may not be part of the MMS. Communication between the SoftSpot Servlets and the MMS may be through XML at the juncture of the SoftSpot Gateway 280 and the Server Gateway 270 shown in Fig. 2.

SOFTSPOT LIFE CYCLE

At the very basic level, the SoftSpot may be represented by a Servlet: the SoftSpotManager. Initially, a Servlet container (for example, Enhydra) spawns a new SoftSpotManager, which in turn spawns an instance of SoftSpotSession for every

Session. SoftSpotSession may maintain one or more protected instances of one or more SoftSpot objects. A SoftSpot may use a descriptor file, for example, SoftSpotDesc, to load, to read, and to store XML directives for the SoftSpot.

Figure 16 is a simplified flow chart illustrating one embodiment of the life cycle of a SoftSpot which may interact with an HDA Server. The following process may repeated for every incoming request. It should be noted that Fig. 16 and the description set forth below is provided by way of example only.

At block 1601, the SoftSpot may receive an HTTPRequest from SoftSpotManager. Upon receipt of the request, the SoftSpot may create a UserAgentInfo object containing data extracted from the http header, and additionally may determine the following data points from the http header: type of User-Agent (WAP browser, Voice browser, HTML browser, and so on); type and size of display; the particular domain from which the request originated; user ID (if applicable); and Session-ID (if applicable). This data extraction is represented at block 1602.

As illustrated at block 1603, the SoftSpot may send an XML Message request to a SoftSpotMMSConnector object, which enables communication with the HDA Server. In one embodiment, an exclusive or a proprietary markup language or Message language may be employed for all internal data transfers on HDA Server, as discussed above (XML or DOM, for example, may be the exclusive language used for internal data communications); in such an embodiment, as shown at block 1604, the SoftSpotMMSConnector may reformat the Message into the proper language (if required) and route it to the MMS. The SoftSpotMMSConnector may next register the

SoftSpot as a SoftSpotMMSResultListener for obtaining results or a reply from the MMS (block 1605).

Responsive to the request, the SoftSpot may receive data returned from the MMS, as shown at block 1606. In one embodiment, the SoftSpot may receive the following information as a result set: seller rating, name, and alias; short and long item descriptions; item images in color and/or in black and white; item and/or deal-type; time remaining for deal completion; communication type (chat, email, telephone, etc.); and the like. The foregoing list is not intended to be exhaustive. It will be appreciated that the information requested or required by the SoftSpot may vary according to application and the context of the interaction.

Upon receiving a result set or a response, the SoftSpot may assign a SoftSpotProcessor object to format the data set into an appropriate format which will be readable by the user-agent or the type of device employed (*e.g.* a desktop computer terminal capable of reading HTML, a hand-held device capable of reading Hand-Held Device Markup Language (HDML), a Web-enabled telephone capable of reading Wireless Markup Language (WML), and so forth). This translation is represented at block 1607. For example, the SoftSpotProcessor may apply rules directed to translating the resulting data (in DOM, for example) as follows: transform raw XML data to domain-specific XML according to Domain Rules; transform domain-styled XML to merchant-styled XML according to Seller Rules; transform merchant-styled XML to user-agent-specific XML depending upon the device used and the connection which is established.

Upon completion of the appropriate translations, the SoftSpot may send the resulting XML Message data to the SoftSpotManager (block 1608), which may pack the XML data into the HTTPResponse stream. When the response is sent, the SoftSpotSession may time out, or alternatively, the user may log out (block 1609). In either case, the SoftSpotManager may be destroyed or may be returned to a pool (block 1610). As discussed above with respect to connection pooling, pooling inactive SoftSpotManager instances may economize on system overhead. The inactive SoftSpotManager may be made active again, to be reused (without having to be recreated) as system load requires.

The SoftSpotManager is the master Servlet controlling specific SoftSpot instances per SoftSpotSession. As discussed above with reference to Fig. 16, as a Servlet, the SoftSpotManager may handle the HTTPRequest and HTTPResult objects, though it may have no knowledge of the content of the streams it handles. As with other elements of the HDA Server, the SoftSpotManager may be load balanced using the various Migration and load balancing schemes discussed above. The SoftSpotManager may have several SoftSpots which it manages in a single SoftSpotSession; in one embodiment, each SoftSpot may be run in its own thread.

Turning now to the attributes of the SoftSpot in particular, it should be appreciated that a SoftSpot is not a Servlet (although it may use a small portion of the Servlet API). Instead, it is "contained" in the SoftSpotManager, which manages it. As noted above, there may be several different SoftSpots per SoftSpotSession.

A SoftSpot generally may maintain a SoftSpotDesc descriptor which holds all the properties and descriptions for the particular SoftSpot. SoftSpots may recognize

some Servlet semantics, such as HTTPRequest and HTTPResult, for example. SoftSpots may spawn threads which govern access to other applications, such as communications consoles and the MMS, for example.

Figure 17 is an illustration of one embodiment of a SoftSpot descriptor. In the exemplary descriptor in Fig. 17, the SoftSpot has received data concerning the following attributes from the http header (HTTPRequest): user-agent; domain type; user-ID; and Session ID. This data extraction is illustrated in context in Fig. 16 at block 1602.

The SoftSpot architecture has a very unique notion of a Session. Specifically, support is provided for simultaneous access by different user-agents, *i.e.* a SoftSpotSession may be joined by another SoftSpotSession which maintains a different SoftSpot.

A Uniform SoftSpot Descriptor (USSD) file is the core of the SoftSpot architecture. The USSD may serve as a master template for the Servlet which controls a specific SoftSpot instance. Each particular SoftSpot may be associated with its own USSD file which controls the overall description of that SoftSpot.

Given a USSD file, a USSD Client (such as a Servlet, for example, the SoftSpotManager discussed above) may find all the relevant transformation instructions and automatically download, cache, and format the markup output (such as HTML). These transformation instructions may be contained in transformation files which may be denoted, for example, by the TXN suffix.

USSD is designed to be extended during runtime. The USSD file may be an XML document that describes a SoftSpot and/or an extension thereof. It is important to

note that the USSD file may not necessarily be an actual file, *per se*. In one embodiment, for example, the data contained in a USSD "file" may simply be stored in the same XML hierarchy in XML or LDAP storage. Figures 18A and 18B are an illustration of one embodiment of a USSD file.

The transformation files for the SoftSpot may be required to be cached locally for the duration of the SoftSpot Session. SoftSpot caching may be provided by the SoftSpotDesc object. At least two types of cache may be available for SoftSpots: in-memory; and file-based. In effect, the main caching of the USSD file and its extensions may be added to the SoftSpotDesc dynamically as needed. As described above, the USSD file's role is to describe where the pertinent transformation file can be downloaded from (if needed) and the order in which the transformation rules are to be applied; the USSD may also define some security information, for example. The SoftSpotDesc may be made aware that a new transform file has been downloaded successfully or, in case of failure, that the required transform file is missing. The SoftSpotDesc may then add the new XML data to its DOM tree (the internal cache).

The SoftSpotDesc may begin to prepare itself immediately upon spawn of a Session, and may in fact run concurrently with the rest of the process. The SoftSpotManager may neither be expected nor required to wait for the SoftSpotDesc for proper operation.

In the case where the SoftSpotDesc fails, a lowest common denominator (*i.e.* a generic, or default, SoftSpotDesc) may be used. Failure can occur in one of the following ways, for example: the SoftSpotDesc cannot download one or more transformation files; the SoftSpotDesc fails to parse one or more transformation files;

the SoftSpotDesc has not completed download of all the required transformation files; or the SoftSpotDesc exits abnormally (VM exit).

The SoftSpotDesc and the SoftSpotManager may be expected to run at least in the same Servlet container (Servlet runner), if not in the same address space. Communication between these objects may be done through the HTTP stream.

From the foregoing, it will be appreciated that an HDA Server system having an HDA Server OS provides a powerful and versatile computer server architecture which overcomes many bandwidth and scalability complications. The preferred embodiments disclosed herein have been described and illustrated by way of example only, and not by way of limitation. Other modifications and variations to the invention will be apparent to those skilled in the art from the foregoing detailed disclosure. While only certain embodiments of the invention have been specifically described herein, it will be apparent that numerous modifications may be made thereto without departing from the spirit and scope of the invention.

WHAT IS CLAIMED IS:

1. A computer server comprising:

a non-hierarchical array of physical machines having physical resources;

a connection allowing data communication between said physical machines; and

program code allocating and managing consumption of said physical resources.

2. The computer server of claim 1 wherein said program code is distributed among two or more of said physical machines.

3. The computer server of claim 1 wherein said program code allows migration of runtime operations across said physical resources.

4. The computer server of claim 1 wherein said program code is an operating system.

5. The computer server of claim 1 wherein said program code defines logical resources comprising at least a portion of said physical resources.

6. The computer server of claim 5 wherein said logical resources are distributed among two or more of said physical machines.

7. The computer server of claim 5 wherein said program code allocates and manages consumption of said logical resources.

8. The computer server of claim 5 wherein said program code is distributed among said logical resources.

9. The computer server of claim 5 wherein said program code allows migration of runtime operations across said logical resources.

10. The computer server of claim 5 wherein said program code migrates across said logical resources.

11. The computer server of claim 5 wherein said program code is an operating system.

12. The computer server of claim 1 further comprising a gateway allowing data communication between said computer server and an external network.

13. The computer server of claim 12 wherein said external network is the internet.

14. The computer server of claim 12 wherein said gateway comprises translation program code translating data between a format used for data communication within said computer server and a plurality of formats used for data communication on said external network.

15. The computer server of claim 12 wherein said data communication between said computer server and said external network is session-based.

16. A computer system comprising:

a distributed computer server comprising a non-hierarchical array of physical machines having physical resources, and a connection allowing data transmission between said physical machines; and

a distributed operating system allocating and managing consumption of said physical resources.

17. The computer system of claim 16 wherein said operating system is distributed among two or more of said physical machines.

18. The computer system of claim 16 wherein said operating system allows migration of runtime operations across said physical resources.

19. The computer system of claim 16 wherein said operating system defines logical resources comprising at least a portion of said physical resources.

20. The computer system of claim 19 wherein said logical resources are distributed among two or more of said physical machines.

21. The computer system of claim 19 wherein said operating system allocates and manages consumption of said logical resources.

22. The computer system of claim 19 wherein said operating system is distributed among said logical resources.

23. The computer system of claim 19 wherein said operating system allows migration of runtime operations across said logical resources.

24. The computer system of claim 19 wherein said operating system migrates across said logical resources.

25. The computer system of claim 16 further comprising a gateway allowing data communication between said distributed computer server and an external network.

26. The computer system of claim 25 wherein said external network is the internet.

27. The computer system of claim 25 wherein said gateway translates data between a format used for data communication within said distributed computer server and a plurality of formats used for data communication on said external network.

28. The computer server of claim 25 wherein said data communication between said distributed computer server and said external network is session-based.

29. The computer system of claim 27 further comprising an adaptive user interface for communicating translated data from said gateway to a destination on said external network in accordance with protocols determined by said destination.

30. The computer system of claim 29 wherein said adaptive user interface is a SoftSpot.

31. A computer system comprising:

a distributed computer server comprising a non-hierarchical array of physical machines having physical resources and a connection allowing data transmission between said physical machines;

an operating system recognizing logical resources; said logical resources being distributed among said physical resources; said operating system managing consumption of said logical resources; and

a gateway allowing data communication between said distributed computer server and an external network.

32. The computer system of claim 31 wherein said operating system is distributed among two or more of said physical machines.

33. The computer system of claim 31 wherein said operating system allows migration of runtime operations across said physical resources.

34. The computer system of claim 31 wherein said operating system is distributed among said logical resources.

35. The computer system of claim 31 wherein said operating system allows migration of runtime operations across said logical resources.

36. The computer system of claim 31 wherein said operating system migrates across said logical resources.

37. The computer system of claim 31 wherein said external network is the internet.

38. The computer system of claim 31 wherein said gateway translates data between a format used for data communication within said distributed computer server and a plurality of formats used for data communication on said external network.

39. The computer server of claim 31 wherein said data communication between said distributed computer server and said external network is session-based.

40. The computer system of claim 38 further comprising an adaptive user interface for communicating translated data from said gateway to a destination on said external network in accordance with protocols determined by said destination.

41. The computer system of claim 40 wherein said adaptive user interface is a SoftSpot.

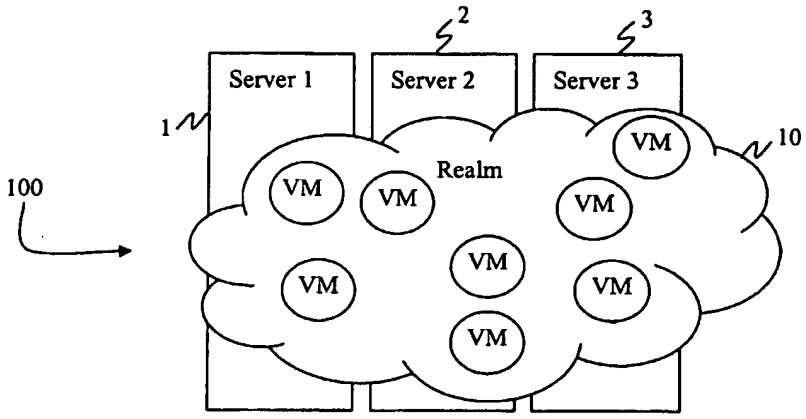

2
3
Server 1
Server 2
Server 3
1
10
100
Realm
VM
VM
VM
VM
VM
VM
VM
VM

FIG. 1

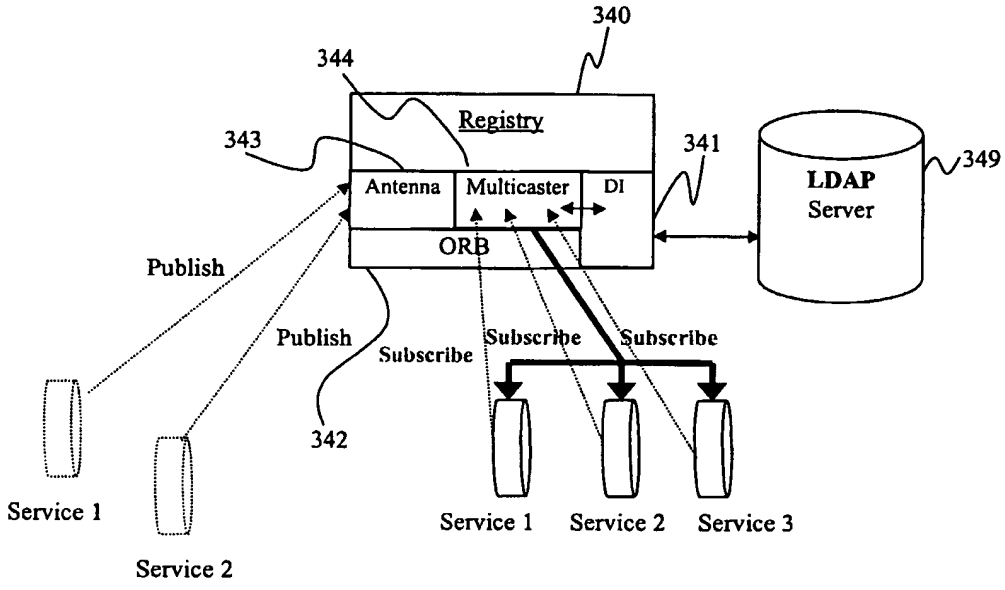

340
344
Registry
343
341
349
Antenna
Multicaster
DI
LDAP
Server
ORB
Publish
Publish
Subscribe
Subscribe
Subscribe
342
Service 1
Service 2
Service 1
Service 2
Service 3

FIG. 3

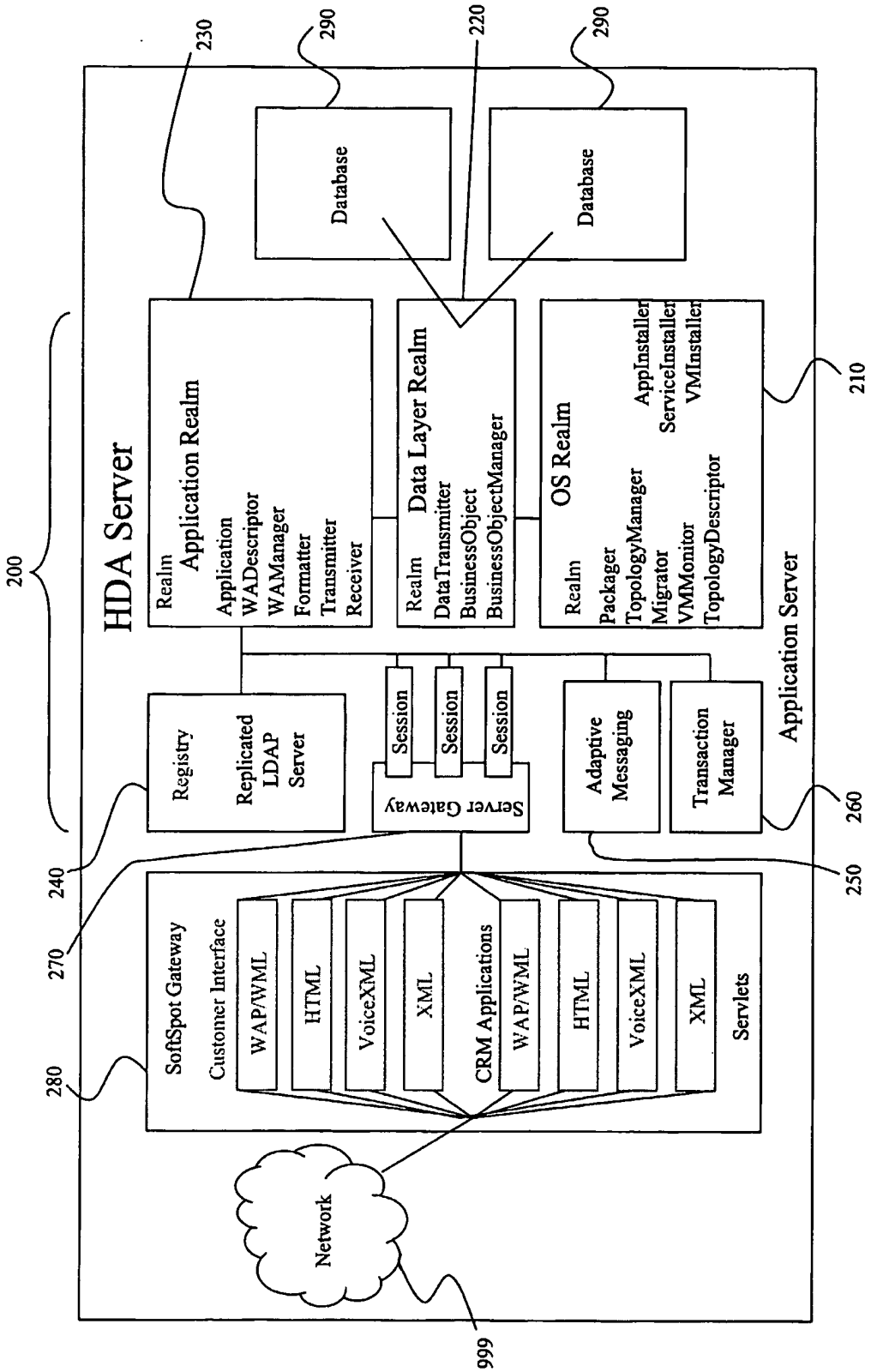

200
HDA Server
230
Realm
Application Realm
Application
WADescriptor
WAManager
Formatter
Transmitter
Receiver
290
Database
220
Realm
Data Layer Realm
DataTransmitter
BusinessObject
BusinessObjectManager
290
Database
Realm
OS Realm
Packager
TopologyManager
Migrator
VMMonitor
TopologyDescriptor
AppInstaller
ServiceInstaller
VMInstaller
210
Application Server
Registry
Replicated
LDAP
Server
240
Session
Session
Session
Server Gateway
270
Adaptive
Messaging
Transaction
Manager
260
250
SoftSpot Gateway
Customer Interface
WAP/WML
HTML
VoiceXML
XML
CRM Applications
WAP/WML
HTML
VoiceXML
XML
Servlets
280
Network
999

FIG. 2

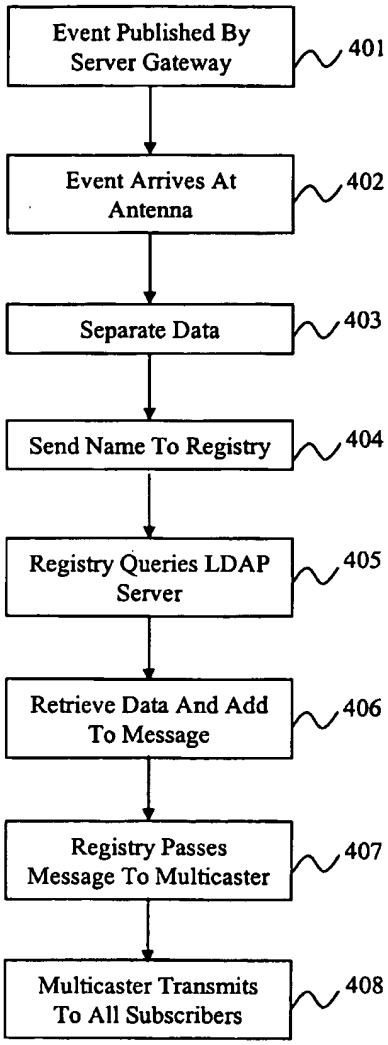

Event Published By Server Gateway
401
Event Arrives At Antenna
402
Separate Data
403
Send Name To Registry
404
Registry Queries LDAP Server
405
Retrieve Data And Add To Message
406
Registry Passes Message To Multicaster
407
Multicaster Transmits To All Subscribers
408

FIG. 4

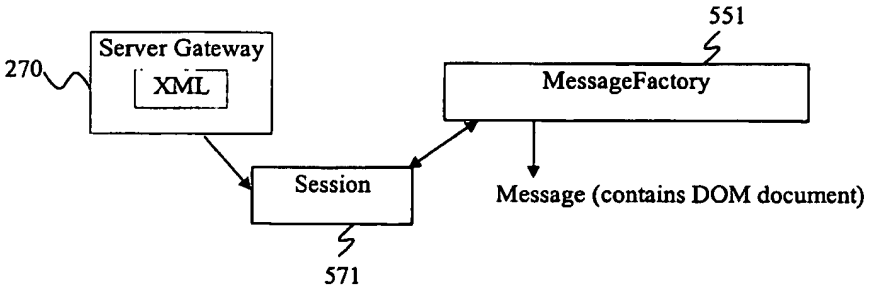

551
Server Gateway
270
XML
MessageFactory
Session
Message (contains DOM document)
571

FIG. 5

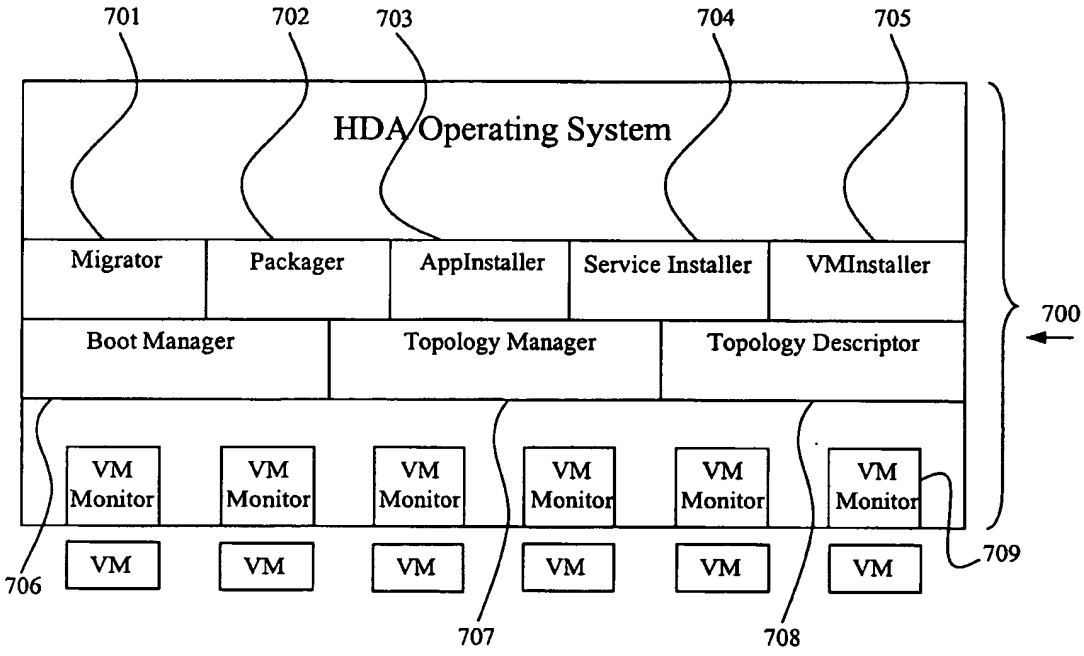

701
702
703
704
705
HDA Operating System
Migrator
Packager
AppInstaller
Service Installer
VMInstaller
Boot Manager
Topology Manager
Topology Descriptor
700
VM Monitor
VM
706
707
708
709

FIG. 7

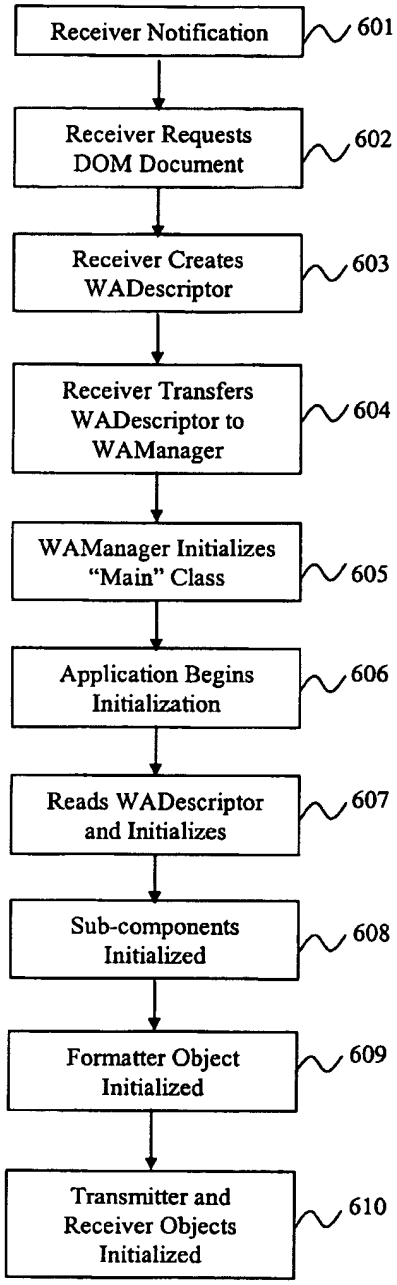

Receiver Notification
601
Receiver Requests DOM Document
602
Receiver Creates WADescriptor
603
Receiver Transfers WADescriptor to WAManager
604
WAManager Initializes "Main" Class
605
Application Begins Initialization
606
Reads WADescriptor and Initializes
607
Sub-components Initialized
608
Formatter Object Initialized
609
Transmitter and Receiver Objects Initialized
610

FIG. 6

```
<topology-descriptor>
    <!- - path to the web application base directory - ->
    <web-application-path base="/usr/local/zebrazone/webapps"/>
    <!- - path to the Plugin base directory - ->
    <Plugin-path base="/usr/local/zebrazone/Plugins"/>
    <!- - path to the resources base directory - ->
    <resource-path base="/usr/local/zebrazone/resources"/>
    <!- - host definition node - ->
    <host type=intel
        os=Linux
        address="111.222.333.444"
        port=1929
        memMeg=512
        host-name="mtTam">
        <!- - vm process definitions (-X type arguments) - ->
        <vm mx=64 jit=none id="f144ca-9ba-f4de-bcb"/>
        <vm mx=64 id="f144ca-9ba-f4de-974"/>
        <vm mx=64 id="b154ca-ffa-f4d5-973"/>
        <vm mx=64 id="f144ca-9ba-f4c-944"/>
    </host>
    <host type=intel
        os=Linux
        address="555.666.777.999"
        port=1929
        memMeg=512
        name="repack">
        <vm mx=64 jit=none id="f144ca-9ba-ffc4-974"/>
        <vm mx=64 id="3efd-9bc-f4de-974"/>
        <vm mx=64 id="fddb-9b4-f4df-cb4"/>
    </host>
    <host type=intel
        os=Linux
```

FIG. 8A

```
            address="31.42.934.733"
            port=1929
            memMeg=512
            name="pineMtn">
            <vm mx=64 id="f144cf-9ba-f4de-974"/>
            <vm mx=64 id="fff4ec-9ba-f4de-974"/>
            <vm mx=64 id="f144ca-9ca-f4de-974"/>
        </host>
        <!- - Realm definitions: name and number of VMs it spans total number for all
            Realms should equal the total number for all hosts - ->
        <Realm name="WebAppplicationRealm" vms number=5 priority=1/>
        <Realm name="DataLayerRealm" vms number=3 priority=2/>
        <Realm name="OSRealm" vms number=2 priority=3/>
        <security>
            <unrestricted/>
        </security>
</topology-descriptor>
```

FIG. 8B

```
<migration-request>
        <target fqn="com.zebrazone.crm.ContactManagement"
                name="Contacts">
                <statistics weightK=13 activity=6>
                <priority>8</priority>
                <reason>load</reason>
        </target>
        . . . etc.
<migration-request>
```

FIG. 9A

```
<?xml version="1.0" encoding="UTF-8"?>
<svcs>
        <registry fqn="com.zebrazone.registry.Registry">
                <ldap-server address="557.799.113.355"
                        port=389
                        searchbase="o=hoo-koo-e-koo"
                        filter="sn=alpine"/> <!- - sample values - - >
        </registry>
<miwok-gateway fqn="com.zebrazone.gateway.MiwokGateway">
</miwok-gateway>
<messaging fqn="com.zebrazone.messaging.MessageFactory">
</messaging>
<Transaction fqn="com.zebrazone.Transaction.TransactionFactory">
</Transaction>
</svcs>
```

FIG. 10

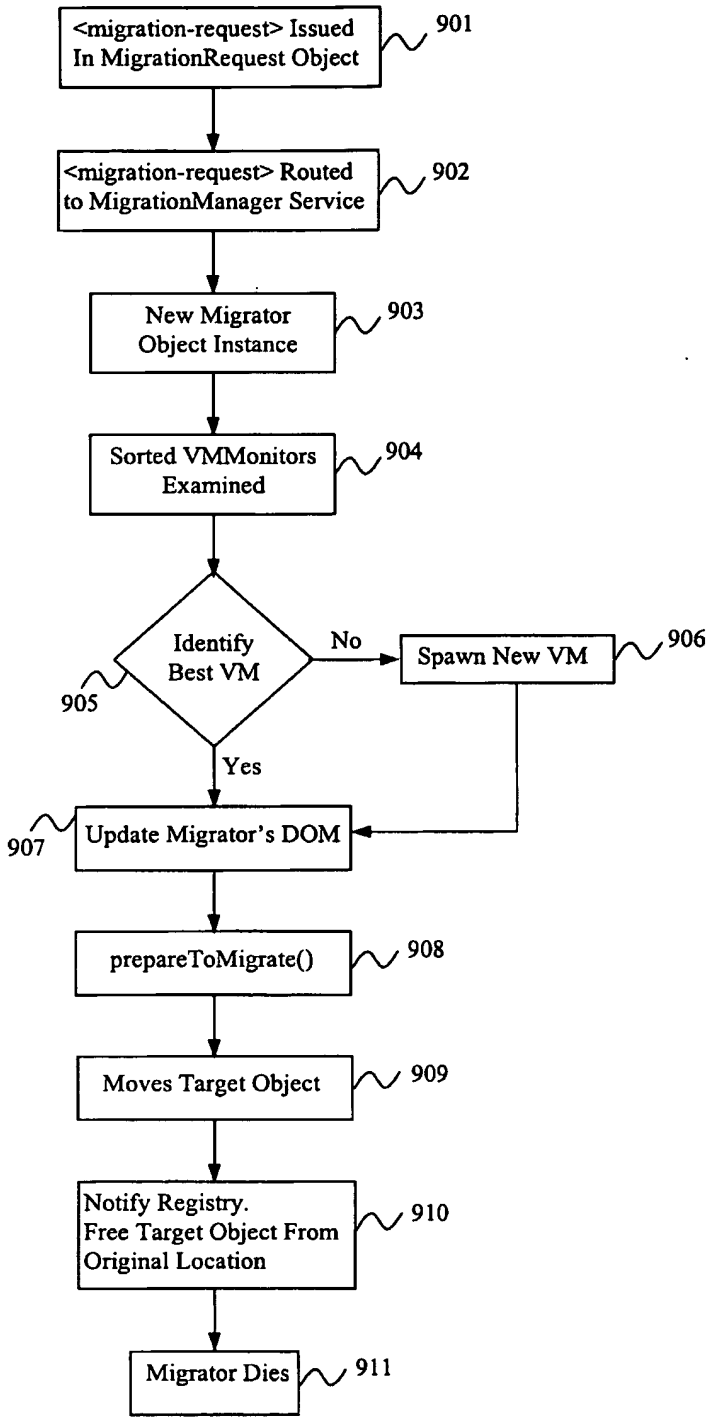

<migration-request> Issued In MigrationRequest Object
901
<migration-request> Routed to MigrationManager Service
902
New Migrator Object Instance
903
Sorted VMMonitors Examined
904
Identify Best VM
905
No
Spawn New VM
906
Yes
Update Migrator's DOM
907
prepareToMigrate()
908
Moves Target Object
909
Notify Registry. Free Target Object From Original Location
910
Migrator Dies
911

FIG. 9B

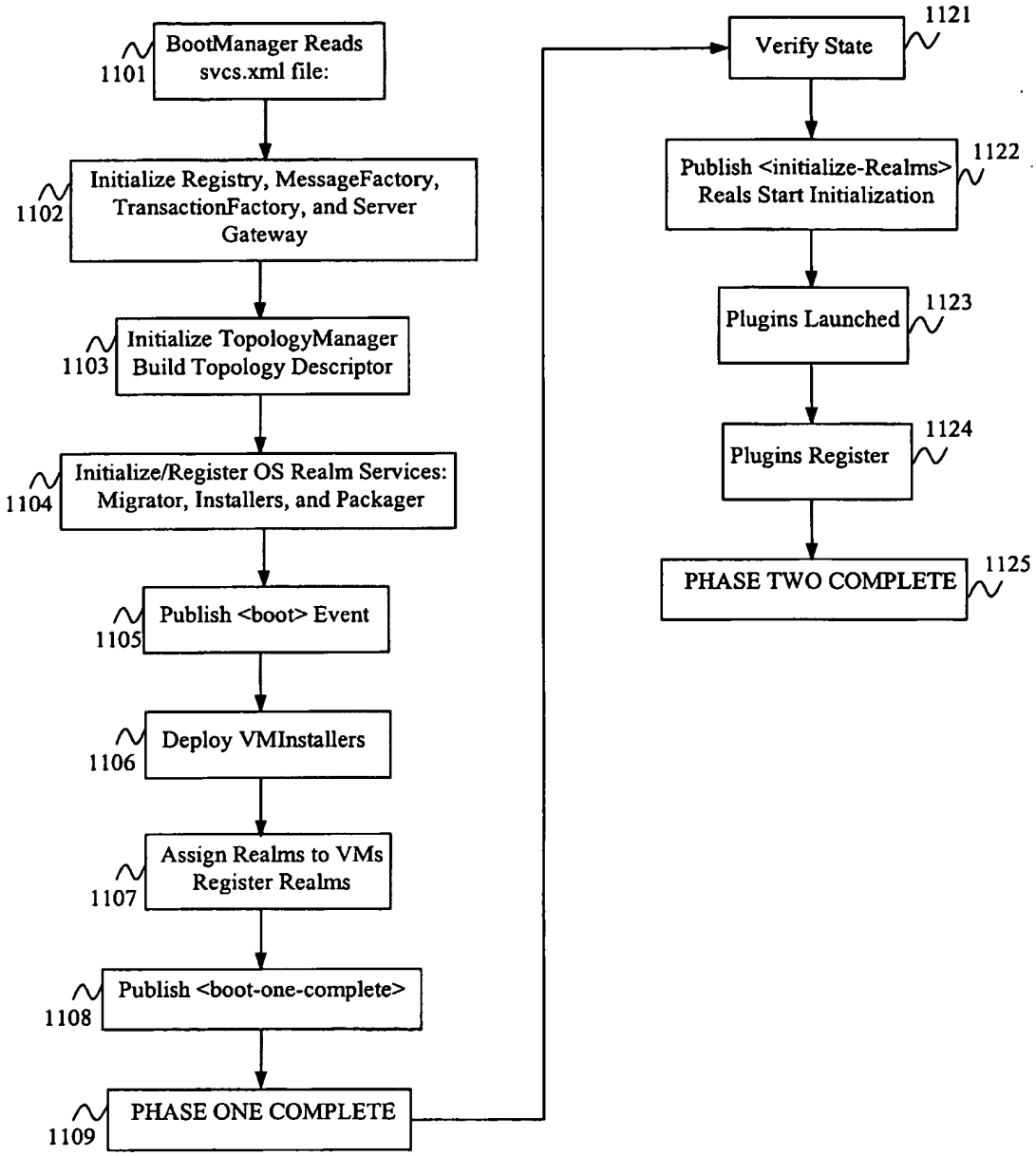

1101
BootManager Reads svcs.xml file:
1102
Initialize Registry, MessageFactory, TransactionFactory, and Server Gateway
1103
Initialize TopologyManager Build Topology Descriptor
1104
Initialize/Register OS Realm Services: Migrator, Installers, and Packager
1105
Publish <boot> Event
1106
Deploy VMInstallers
1107
Assign Realms to VMs Register Realms
1108
Publish <boot-one-complete>
1109
PHASE ONE COMPLETE
1121
Verify State
1122
Publish <initialize-Realms> Reals Start Initialization
1123
Plugins Launched
1124
Plugins Register
1125
PHASE TWO COMPLETE

FIG. 11

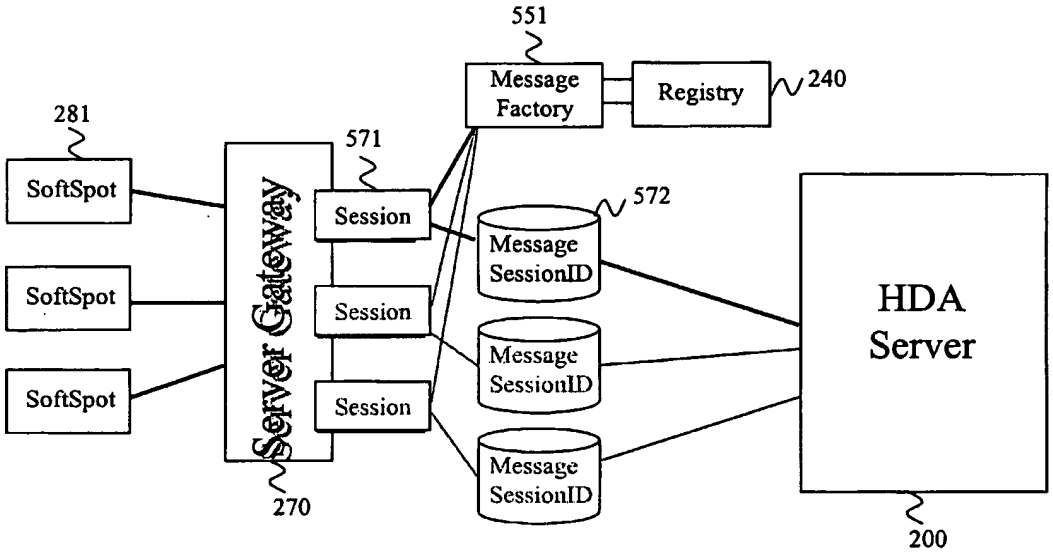

551
Message Factory
Registry
240
281
571
SoftSpot
SoftSpot
SoftSpot
Server Gateway
Session
Session
Session
572
Message SessionID
Message SessionID
Message SessionID
HDA Server
270
200

FIG. 12

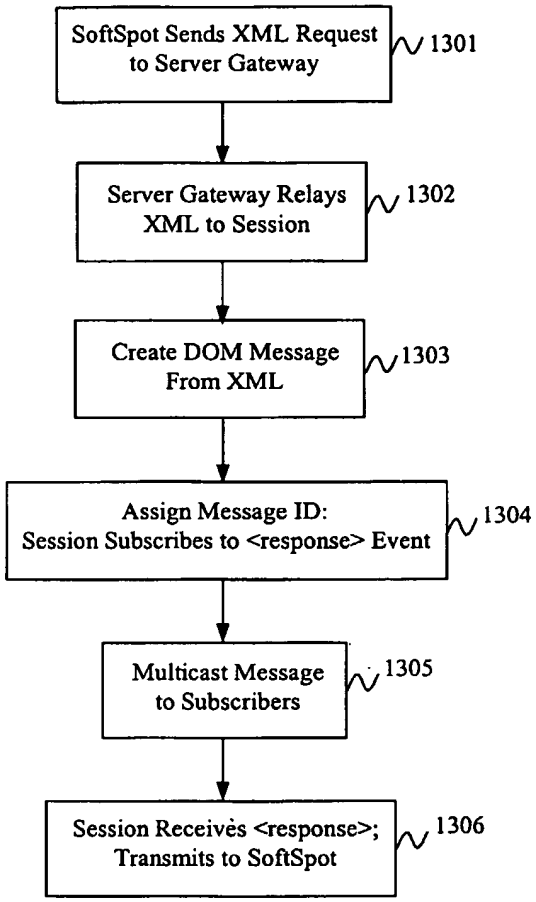

SoftSpot Sends XML Request to Server Gateway
1301
Server Gateway Relays XML to Session
1302
Create DOM Message From XML
1303
Assign Message ID: Session Subscribes to <response> Event
1304
Multicast Message to Subscribers
1305
Session Receives <response>; Transmits to SoftSpot
1306

FIG. 13

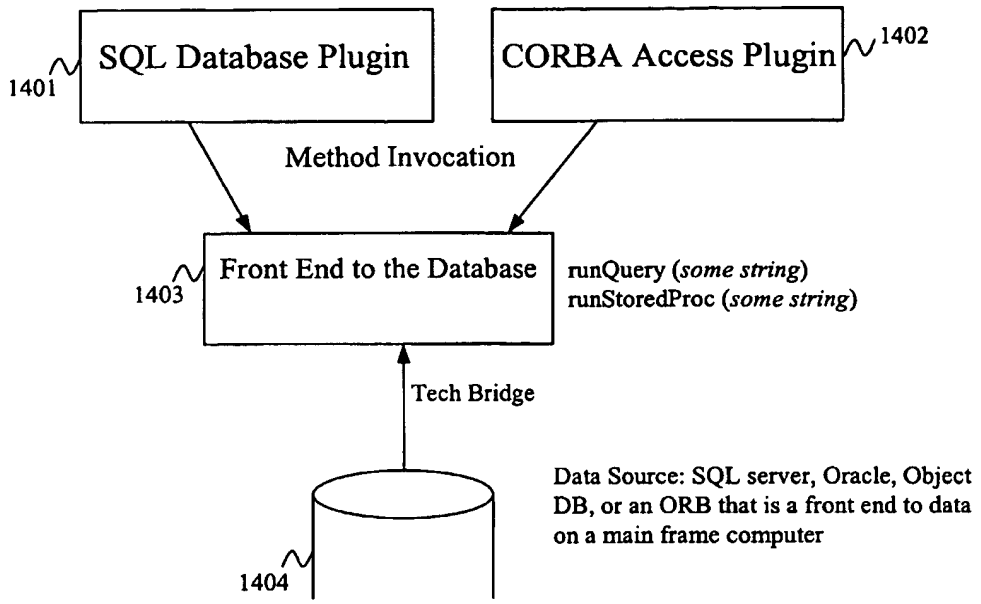

SQL Database Plugin
1401
CORBA Access Plugin
1402
Method Invocation
Front End to the Database
1403
runQuery (some string)
runStoredProc (some string)
Tech Bridge
Data Source: SQL server, Oracle, Object DB, or an ORB that is a front end to data on a main frame computer
1404

FIG. 14

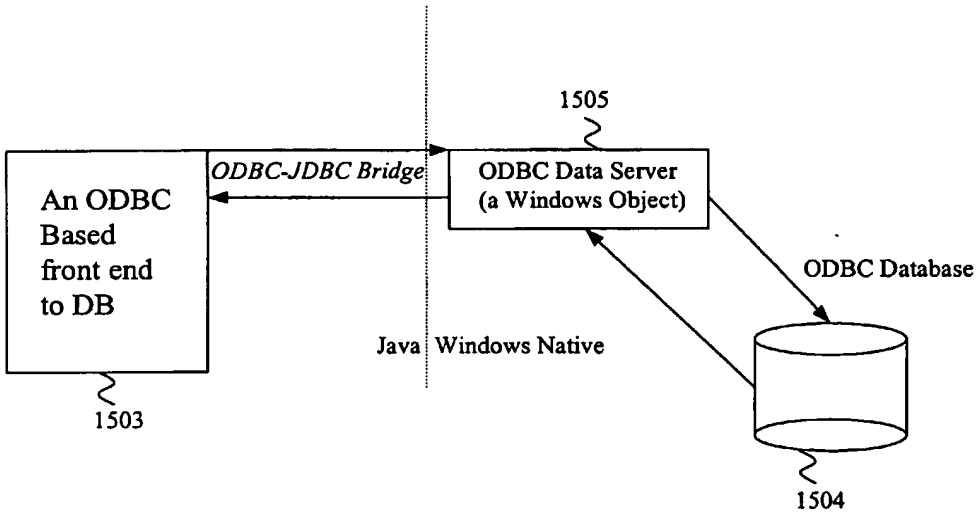

1505
ODBC-JDBC Bridge
ODBC Data Server
(a Windows Object)
An ODBC Based front end to DB
ODBC Database
Java
Windows Native
1503
1504

FIG. 15

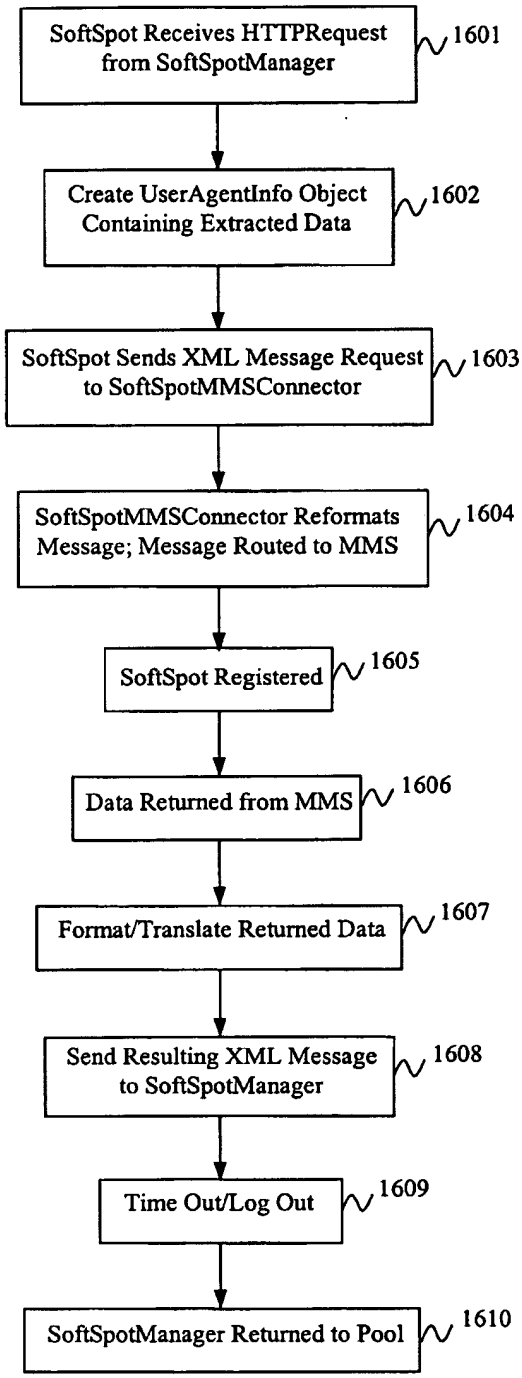

SoftSpot Receives HTTPRequest from SoftSpotManager
1601
Create UserAgentInfo Object Containing Extracted Data
1602
SoftSpot Sends XML Message Request to SoftSpotMMSConnector
1603
SoftSpotMMSConnector Reformats Message; Message Routed to MMS
1604
SoftSpot Registered
1605
Data Returned from MMS
1606
Format/Translate Returned Data
1607
Send Resulting XML Message to SoftSpotManager
1608
Time Out/Log Out
1609
SoftSpotManager Returned to Pool
1610

FIG. 16

```
<?xml version="1.0" encoding="UTF-8"?>
<zzml>
    <message>
        <class>
            <client-information/>
        </class>
        <header>
            <security>
                <unrestricted/>
            </security>
            <priority>
                <default/>
            </priority>
        </header>
        <body>
            <user-agent>
                <type>WAP</type>
                <language>WML</language>
                <version>1.0</version>
            </user-agent>
            <domain>
                <base> DomainNameGoesHere </base>
                <branch>stores</branch>
            </domain>
            <user-id>14CB9F9045</user-id>
            <Session-id>1E489-27C-1903-2ADCF</Session-id>
        </body>
    </message>
</zzml>
```

FIG. 17

```
<?xml version="1.0" encoding="UTF-8"?>
<ussd codebase="http://www.abc-site.com/zzone/servlet>
        <order>
                <default/>
        </order>
        <domain-information>
                <master>
                        <name>DomainNameGoesHere</name>
                        <security>
                                <key>2490FC0BC87DA</key>
                                <store-pass>th3W0r1d</store-pass>
                                <alias>boohoo132</alias>
                        </security>
                        <domain-transform-file codebase="ldap://domainname:main.txn"/>
                </master>
                <sub>
                        <name>Stores</name>
                        <security>
                                <key>2490FC0BC87DA</key>
                                <store-pass>th3W0r1d</store-pass>
                                <alias>boohoo132</alias>
                        </security>
                        <domain-transform-file
                codebase="ldap://domainname/stores:main.txn"/>
                </sub>
        </domain-information>
        <seller-information>
                <seller-id>14914890</seller-id>
                <seller-transform-file codebase=ldap://14914890:prefs.txn>
```

FIG. 18A

```
        </seller-information>
        <user-agent-information>
                <wap codebase=ldap://....txn/>
                <html codebase=ldap://....txn/>
                <pqa codebase=ldap://....txn/>
                <vxml codebase=ldap://....txn/>
                <xhtml codebase=ldap://....txn/>
                <applet codebase=ldap://....txn/>
                <flash codebase=ldap://....txn/>
                ...
        </user-agent -information>
        <security>
                <unrestricted/>
        </security>
        <extension codebase=http://....txn/>
        <extension codebase=http://....txn/>
</ussd>
```

FIG. 18B

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/31108

A. CLASSIFICATION OF SUBJECT MATTER
IPC(7) : G06F 15/16, 13/00; G01B 7/00; H04N 7/10; H04J 3/02
US CL : 709/227; 395/ 200.13, 200.76, 800; 364/551.02; 370/455
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
U.S. : 709/227; 395/ 200.13, 200.76, 800; 364/551.02; 345/327

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y, P	US 6,085,247 A (PARSONS et al.) 04 July 2000 (04.07.2000), see columns 4-6.	1-41
Y	US 5,604,867 A (HARWOOD) 18 February 1997 (18.02.1997), see columns 9-12.	1-41
Y	US 5,655,140 A (HADDOCK) 05 August 1997 (05.08.1997), see Summary of the Invention.	1-41
Y, P	US 6,061,360 A (MILLER et al.) 09 May 2000 (09.05.2000), see columns 5-7.	1-41
Y	US 5,556,092 A (WANG et al.) 15 October 1996 (15.10.1996), see columns 10-13.	1-41
Y	US 5,560,038 A (HADDOCK) 24 September 1996 (24.09.1996), see Summary of the Disclosure.	1-41

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
26 December 2000 (26.12.2000)	25 JAN 2001
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. 703 305-3230	Authorized officer ALVIN OBERLEY James R. Matthews Telephone No. 703 305-3665

Form PCT/ISA/210 (second sheet) (July 1998)